НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Факультет інформатики та обчислювальної техніки
(повна назва інституту/факультету)

Автоматизованих систем обробки інформації і управління
(повна назва кафедри)

«На правах рукопису»
УДК 004.43

До захисту допущено:
В.о. завідувача кафедри
_____ Олександр ПАВЛОВ
«___»_____20__ р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-професійною програмою «Інженерія програмного забезпечення комп'ютеризованих систем»

зі спеціальності 121 «Інженерія програмного забезпечення»

на тему: «Математичне та програмне забезпечення для визначення характерних рис програмного коду»

Виконав (-ла):
студент (-ка) VI курсу, групи ІП-92мп
Ващенко Юрій Олександрович _____

Науковий керівник:
Проф., д.т.н.,
Фіногенов Олексій Дмитрович _____

Рецензент:
Доцент, д.т.н,
Корнага Ярослав Ігорович _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент (-ка) _____

Київ – 2020 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Автоматизованих систем обробки інформації і управління

Рівень вищої освіти – другий (магістерський)

Спеціальність – 121 «Інженерія програмного забезпечення»

Освітньо-професійна програма - «Інженерія програмного забезпечення комп'ютеризованих систем»

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Олександр ПАВЛОВ
«___»_____20__ р.

ЗАВДАННЯ
на магістерську дисертацію студенту
Ващенку Юрію Олександровичу

1. Тема дисертації «Математичне та програмне забезпечення для визначення характерних рис програмного коду», науковий керівник дисертації Фіногенов Олексій Дмитрович, проф., д.т.н., затверджені наказом по університету від «26» жовтня 2020 р. *№ 3132-с*

2. Термін подання студентом дисертації _____

3. Об'єкт дослідження процес статичного аналізу вихідного коду програм.

4. Вхідні дані Технічне завдання

5. Перелік завдань, які потрібно розробити дослідити методи статичного обробки та аналізу коду; виконати огляд існуючих інструментів, що виконують перевірку коду; вибрати алгоритми, що можуть стати основою для покращення; розробити алгоритм, що забезпечує збільшення інформації про код в процесі запуску; розробити програмну реалізацію алгоритму; провести порівняння якості отриманих результатів з існуючими відкритими аналогами; провести аналіз отриманих результатів.

6. Орієнтовний перелік графічного (ілюстративного) матеріалу

1) Схема структурна класів програмного забезпечення

7. Орієнтовний перелік публікацій 1 публікація

8. Консультанти розділів дисертації

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Графічний	доц. Ліщук К.І.		

9. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	*Вивчення рекомендованої літератури*	*06.09.2020*	
2	*Аналіз метрик нормальної роботи суден*	*06.09.2020*	
3	*Аналіз підходів до архітектури*	*07.09.2020*	
4	*Постановка та формалізація задачі*	*12.09.2020*	
5	*Дослідження існуючих алгоритмів для побудови метрик*	*22.09.2020*	
6	*Моделювання програмного забезпечення*	*20.10.2020*	
7	*Розробка програмного забезпечення*	*01.11.2020*	
8	*Виконання графічних документів*	*25.11.2020*	
9	*Оформлення дисертації*	*25.11.2020*	
10	*Подання МД на попередній захист*	*27.11.2020*	
11	*Подання МД рецензенту*		
12	*Подання МД на основний захист*	*18.12.2020*	

Студент Ващенко Ю.О.

Науковий керівник Фіногенов О.Д.

РЕФЕРАТ

Актуальність. Безперервний розвиток комп’ютерних систем призводить до все більшох кількості створюваних програмних продуктів, до того ж, у зв’язку з постійно зростаючими вимогами, виростає складність існуючого програмного коду. Як результат маємо дуже складу систему програмних продуктів, які, до того ж, пов’язані між собою. Одни із наслідків цього є поява великої кількості помилок в програмному забезпеченні, а тоже і до зменшення показника стабільності. Це є однією з найбільших проблем оскільки сповільнює зріст програмного середовища, та найгірше, може поставити крапку на напрямках пов’язаних з аналітикою та прияняттям рішень, систем реального часу, медичних системах. Тому все більше та більше актуальним стають засоби для аналізу якості програмних продуктів. На даний момент вже є велика різноманітність засобів, крім того, й інструментів їх інтеграції в процеси навчання та розробки код. На жаль, більша частина із цих засобів виконує лише поверхневу перевірку, та не включає складні випадки, такі як прогнозування можливих шляхів виконання коду, аналіз помилок пов’язаних зі станом програми. Ось чому засоби для статичного аналізу є актуальними назараз та їх актуальність лише буде збільшуватись із часом.

Зв'язок роботи з науковими програмами, планами, темами. Робота виконувалась на кафедрі автоматизованих систем обробки інформації та управління Національного технічного університету України «Київський політехнічний інститут ім. Ігоря Сікорського» в рамках ініціативної теми «Інтелектуальні методи програмування, моделювання і прогнозування з використанням ймовірністного і лінгвістичних підходів».

Мета дослідження – покращення процесу аналізу вихідного коду, розробка алгоритму, що виконує більш глибокий типізований аналіз програмного коду.

Для досягнення поставленої мети необхідно виконати наступні **завдання**:

– дослідити методи статичного обробки та аналізу коду;

– виконати огляд існуючих інструментів, що виконують перевірку коду;

– вибрати алгоритми, що можуть стати основою для покращення;

– розробити алгоритм, що забезпечує збільшення інформації про код в процесі запуску;

– розробити програмну реалізацію алгоритму;

– провести порівняння якості отриманих результатів з існуючими відкритими аналогами;

– провести аналіз отриманих результатів.

Об’єкт дослідження – є процес статичного аналізу вихідного коду програм.

Предмет дослідження – є алгоритми статичного аналізу синтаксичної та семантичної моделі програмного коду.

Наукова новизна отриманих результатів полягає у отриманні методу аналізу статичної та семантичної моделі коду на надійність на основі модифікованого алгоритму виводу типів Хіндлі-Мілнера, що забезпечує отримання більш детальної інформації про стан вихідного коду та дозволяє підвищити його якість.

Практичне значення одержаних результатів отриманих в роботі результатів полягає в тому, що розроблений спосіб верифікації програмного забезпечення надає більш детальну інформацію, щодо якості вихідного коду, наприклад, об'єктів, що мають змінюваний стан та є поліморфними та може використовуватись в процесі розробоки програмного забезпечення в якості верифікатора та в процессі в навчать в якості контролю.

Публікації. Результати проведених досліджень були опубліковані в дисертації «Застосування статичного аналізу коду на прикладі Data-flow аналізу» рамках всеукраїнської науково-практичної конференції молодих вчених та студентів «Інформаційні системи та технології управління» (ІСТУ-2020) -м Київ: НТУУ «КПІ ім. Ігоря Сікорського».

СТАТИЧНИЙ АНАЛІЗ КОДУ, ШАБЛОНИ ПРОЕКТУВАННЯ, ПОТІК ДАНИХ, ФУНКЦІОНАЛЬНЕ ПРОГРАМУВАННЯ

ABSTRACT

Topicality. The continuous development of computer systems leads to an increasing number of software products created, in addition, due to the ever-increasing requirements, the complexity of the existing software code is growing. As a result, we have a very complex system of software products, which, moreover, are interconnected. One of the consequences of this is the appearance of a large number of errors in the software, as well as a decrease in stability. This is one of the biggest problems because it slows down the growth of the software environment, and worst of all, it can put an end to areas related to analytics and decision making, real-time systems, medical systems. Therefore, more and more relevant tools for analysing the quality of software products. At the moment, there is already a great variety of tools, in addition, and tools for their integration into learning and code development. Unfortunately, most of these tools perform only a superficial check, and do not include complex cases, such as predicting possible ways to execute code, analysing errors related to the state of the program. That is why the tools for static analysis are relevant now and their relevance will only increase over time.

Connection of work with scientific programs, plans, themes. The work was performed at the Department of Automated Information Processing and Control Systems of the National Technical University of Ukraine "Kyiv Polytechnic Institute. Igor Sikorsky ”in the framework of the initiative topic“ Intelligent methods of programming, modeling and forecasting using probabilistic and linguistic approaches ”.

The purpose of the study is to improve the process of source code analysis, to develop an algorithm that performs a deeper typed analysis of program code.

To achieve this goal it is necessary to perform the following tasks:

- explore methods of static code processing and analysis;
- review existing tools that perform code validation;
- choose algorithms that can be the basis for improvement;
- based on the analysis to perform an algorithm that gives more data about the code during startup;

- develop a software implementation of the algorithm;
- to compare the quality of the obtained results with the existing open analogues;
- to analyze the obtained results.

The object of research is the process of static analysis of program source code.

The subject of research is algorithms of static analysis of syntactic and semantic model of program code.

The scientific novelty of the obtained results is to obtain a method of static analysis of the syntactic and semantic model of the source code for reliability based on the Hindley-Milner type inference algorithm.

The practical significance of the results obtained in the work is that the developed method of software verification provides more detailed information on the quality of the source code, for example, objects that have a variable state and are polymorphic and can be used in software development in as a verifier and in the process of being taught as a control.

Publications. The results of the research were published in the dissertation "Application of static code analysis on the example of Data-flow analysis" in the framework of the All-Ukrainian scientific-practical conference of young scientists and students "Information systems and control technologies" (ISTU-2020) - Kyiv: NTUU " Igor Sikorsky Kyiv Polytechnic Institute ».

STATIC CODE ANALYSIS, DESIGN TEMPLATES, DATA FLOW, FUNCTIONAL PROGRAMMING

ЗМІСТ

ВСТУП **11**
1 ЗАГАЛЬНІ ПОЛОЖЕННЯ **12**
1.1 Опис предметного середовища 12
1.1.1 Опис процесу діяльності 12
1.1.2 Опис функціональної моделі 13
1.2 Огляд наявних аналогів 13
1.3 Аналіз вимог до програмного забезпечення 15
1.3.1 Варіанти використання 15
1.4 Постановка задач дослідження 17
Висновок до розділу 17
2 ТЕОРЕТИЧНІ ОСНОВИ **19**
2.1 Абстрактне синтаксичне дерево 19
2.2 Семантична модель коду 21
2.3 Інкапсуляція 22
2.4 Поліморфізм 23
2.5 Алгоритм Хіндлі-Мілнера для виводу типів 23
Висновок до розділу 26
3 АНАЛІЗ МЕТОДІВ СТАТИЧНОГО АНАЛІЗУ КОДУ **27**
3.1 Data-flow аналіз 27
3.2 Аналіз контрольної залежності 31
3.3 Taint аналіз 32
3.4 Структурний аналіз в мові Typescript 33
3.5 Аналіз методів статичного аналізу коду 34
Висновки до розділу 36
4 ОПИС ПРОГРАМНОГО ПРОДУКТУ **37**
4.1 Архітектура програмного забезпечення 37

4.2 Засоби розробки 37
4.2.1 Roslyn 38
4.2.2 MongoDB 39
4.2.3 Fastify 42
4.3 Вимоги до технічного забезпечення 43
4.4 Опис архітектури програмного забезпечення 44
4.4.1 Опис класів 44
4.3.2 Опис методів 48
4.3.3 Діаграма компонентів 54
Висновки до розділу 55
5 МОДИФІКОВАНИЙ АЛГОРИТМ СТАТИЧНОГО АНАЛІЗУ 56
5.1 Опис розроблюваного алгоритму 56
5.2 Розроблена структура даних для оперування 58
5.3 Порівняльний аналіз 61
5.4 Обчислювальна складність алгоритму 62
Висновок до розділу 64
6 РОЗРОБЛЕННЯ СТАРТАП ПРОЕКТУ 65
6.1 Опис ідеї проекту 65
6.2 Технічний аудит проєкту 67
6.3 Аналіз ринку для запуску проєкту 67
6.4 Маркетингова програма стартап-проєкту 68
6.5 Ринкова стратегія проєкту 70
Висновки до розділу 72
ВИСНОВКИ 74
ПЕРЕЛІК ПОСИЛАНЬ 76

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

AST – abstract syntax tree.

API – application programming interface.

JSON – JavaScript Object Notation.

SOLID – набір з 5 правил, що повинні виконуватись будь яким розробником при кодуванні.

REST – Representational State Transfer, система правил для побудови API

SQL – мова запитів до реляційної бази даних.

SQL-ін’єкція – метод атаки на реляційну базу даних.

Statefull-об’єкті – об’єкти, дії яких залежать від поточного стану, який є змінюваним.

ООП – об’єктно-орієнтоване програмування.

CDG – граф залежних умов.

CFG – граф потоку управління.

MVC – паттерн на основі компонентів model(модель), view(представлення), controller(контроллер).

ВСТУП

Етап верифікації програмного забезпечення включає в себе складну систему обробки вихідного коду та сможе бути виконаний великою кількістю способів – це і синтаксичний і семантичний аналіз, також перевірка на наявність неактивних блоків коду, умов, що ніколи не виконується, повторення, помилки орфографії. Цій темі посвячена невелика кількість наукових робіт, оскільки ця тема сама по собі є допоміжною в розробці, а не виконання окремих функціональних блоків.

Існуючі накові роботи впроваджують процеси перевірки безпеки, зокрема використання API, SQL-ін’єкції, пошук паттернів, і тд. Проте наразі існує досить невелика кількість робіт стосовно аналізу потоку вихідного коду[1-5]. Одне із не вирішених питань є аналіз поліморфного коду, а також коду, що пов’язаний зі змінюваним станом об’єктів. Тому одною з головних цілей цієї роботи є подолання цього бар’єру для отримання більш якісної картини по коду а також перевірка можливості використання алгоритмів виводу типів з функціональних мов програмування до імперативного коду. Другорядними елементами роботи аналіз рис програмного коду, зокрема цикломатичної складності та пошук використаних шаблонів програмування. Разом цей функціонал складає комплексну систему верифікації та виведення рис з програмного коду. Та по своїй суті складає невирішену проблему повного статичного аналізу, проте, з іншої сторони є лише частковим підходом в системі в обробки вихідного коду.

Дане дослідження і розробка алгоритмічного забезпечення є актуальним для команд інженерної розробки та в навчальному процесі. Потреба в більш якісному виконання етапу верифікації зростає, особоливо це помітно в нових галузях, що пов’язані з прийняттям рішень та систем реального часу, а отже і зростають запити до систем статичного аналізу.

1 ЗАГАЛЬНІ ПОЛОЖЕННЯ

1.1 Опис предметного середовища

Статичний аналіз коду - це сукупність засобів для отримання інформації про якість, структуру, особливості виконання, надійність програмного кода[1]. Він дозволяє спростити аналіз програмного коду та може бути корисним для розробників з метою покращення якості та пошуку вад в програмному коді, замовників - з метою отримання інформації про стан проекту, його якість. Керівників IT-підприємств чи відділів для системного аналізу діяльності. Визначення характерних рис програмного - це процес проведення статичного аналізу, а також при необхідності і додаткових дій - наприклад консультування, пошуку плагіатів тощо.

1.1.1 Опис процесу діяльності

Оскільки основною ціллю статичного аналізу коду є аналіз коду, то процесам при яких він застосовується є процеси, що лежать в основі практик розробки та розгортування програмного коду, зокрема при Continuous Integration (CI) в якості обмежувача або ж просто для консультування. Також він може виконуватись в процесі отримання звітів чи прийомів лабораторних чи інших робіт в учбових закладах, проведення змагань. До введення автоматизації в рамках дисертації процес CI виглядав як операції над системою контролю версій, тестуванням, генерацією додаткових проектних файлів за необхідності. Основними способами зупинити цей процес були помилки компіляції, помилки при запусках тестів, проблеми з мережею, файловим середовищем. Ці способи не надають можливість зупинити процес інтеграції коду в тому разі, якщо він не відповідає показникам якості, окрім проходження тестів. В цьому і відмінність розроблюваної системи аналізу коду від процесів діяльності, що були раніше. Аналогічно і для інших способів використання: роботи по програмах навчальних закладів, змагання, - крім звітів, в разі яких визначення характерних рис програмного коду дозволяє всього лише отримати дані про базу коду, динаміку змін в коді.

1.1.2 Опис функціональної моделі

В рамках використання програмного забезпечення для визначення характерних рис програмного коду будуть задіяні такі актори:

- автор коду;
- людина, яка хоче перевірити якість коду або отримати її показники;
- автоматизована система для CI.

В окремих випадках людиною, що перевіряє код може бути сам автор коду.

Згідно даного списку дійових осіб та призначення програмного забезпечення можна виділити такі варіанти використання, що будуть притаманні проекту:

- зупинка процесу інтеграції коду в репозиторій;
- перевірка коду на якість в навчальних закладах;
- перевірка коду на якість та інші показники при організації змагань;
- отримання звітності про стан та динаміку коду в IT-проекті.

1.2 Огляд наявних аналогів

Серед наявних аналогів можна виділити такі програмні комплекси:

- SonarQube;
- PVS-Studio.

SonarQube – є комплексною системою для отримання оцінки якості програмного коду, він має багато інтеграцій, дозволяє робити звіти. Активно використовується при розробці програмного забезпечення. Основними недоліками є відсутність глибоко аналізу на вади програмного коду, не найкраща якість роботи з окремими технологіями, для яких більш специфічні інструменти можуть давати кращі показники. На рисунку 1.1 зображено користувацький інтерфейс.

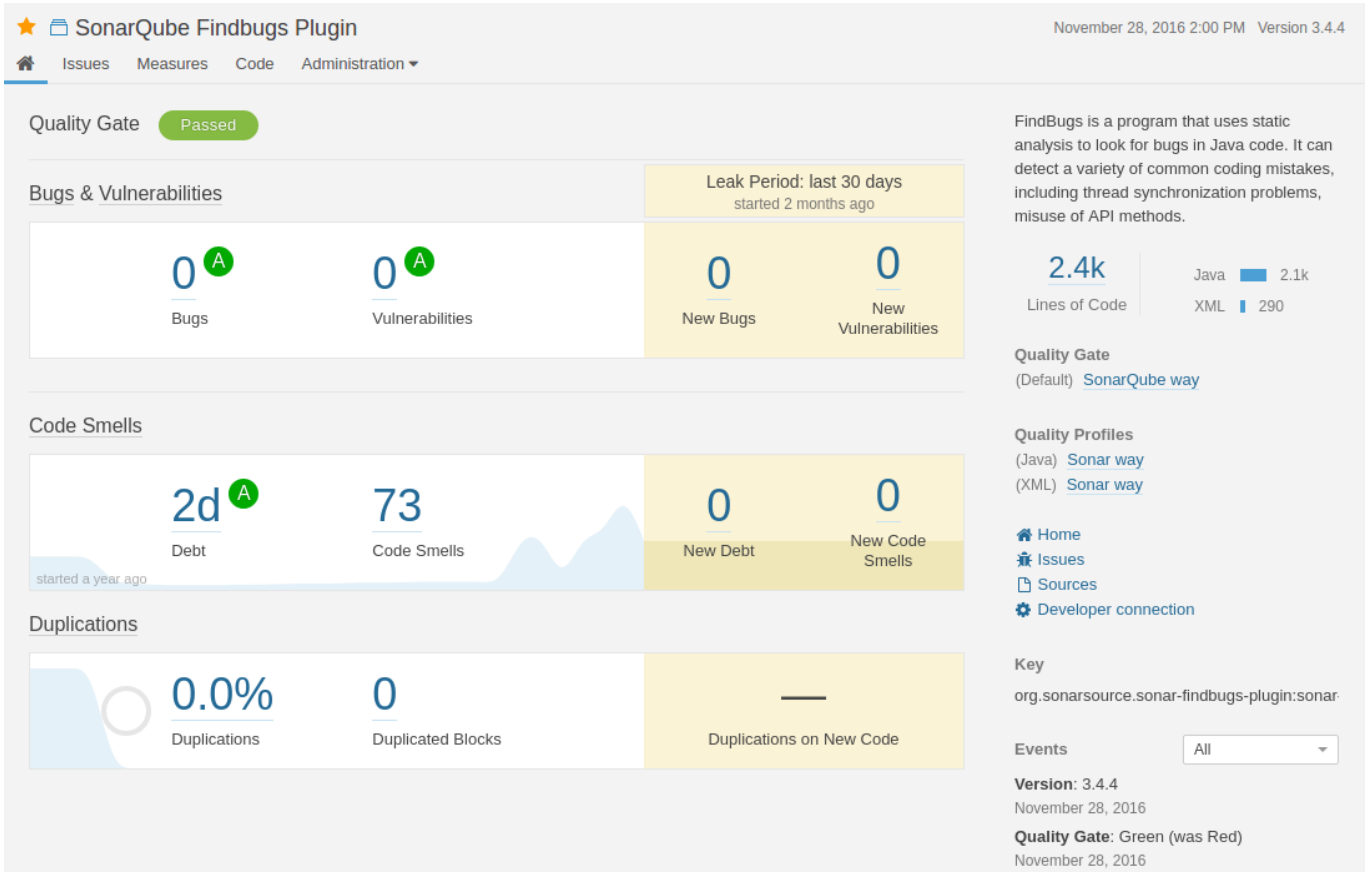


Рисунок 1.1 – Приклад інтерфейсу SonarQube

Roslyn – система статичного аналізу та компілятор для мови C#, дозволяє дізнатись велику кількість показників для коду на цій мові, є основою екосистеми .NET.

PVS-Studio – система аналізу якості та пошуку вад в програмному забезпеченні, проводить найглибший аналіз з існуючих аналогів, проте вона є комерційною та досить недешевою. На рисунку 1.2 зображено приклад користувацького інтерфейсу PVS-studio плагіну до IDEA.

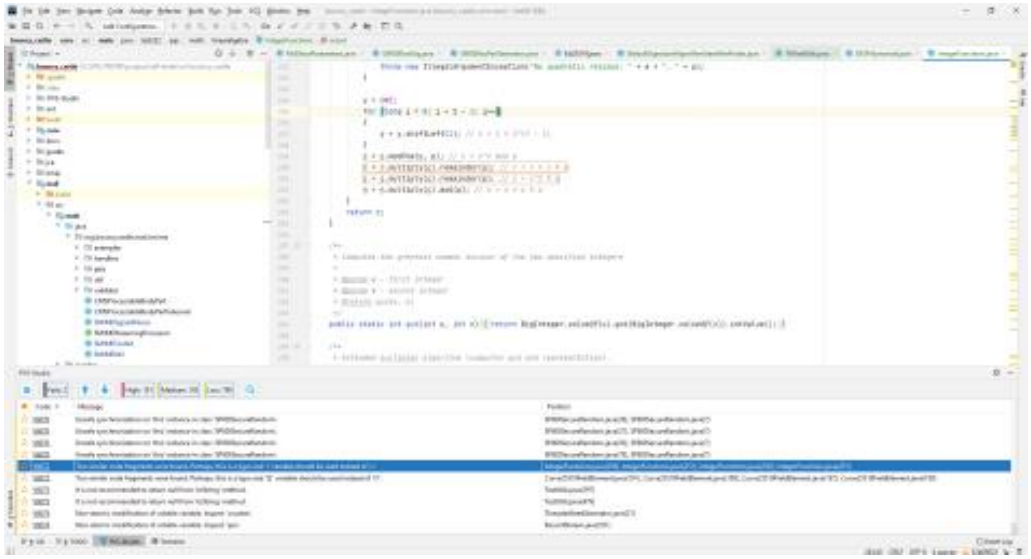


Рисунок 1.2 – Приклад інтерфейсу PVS-Studio

Як бачимо, інтегрований інтерфейс PVS-Studio містить функціональність для пошуку помилок в окремому вікні в інтерфейсі.

1.3 Аналіз вимог до програмного забезпечення

1.3.1 Варіанти використання

Проаналізуємо варіанти використання системи в різних умовах. Даний проєкт буде використовуватись в 2 видах робіт:

– навчальний процес;

– розробка програмного забезпечення.

Кожен процес має свій набір дійових осіб. Зобразимо варіанти використання на рисунках 1.3 та 1.4.

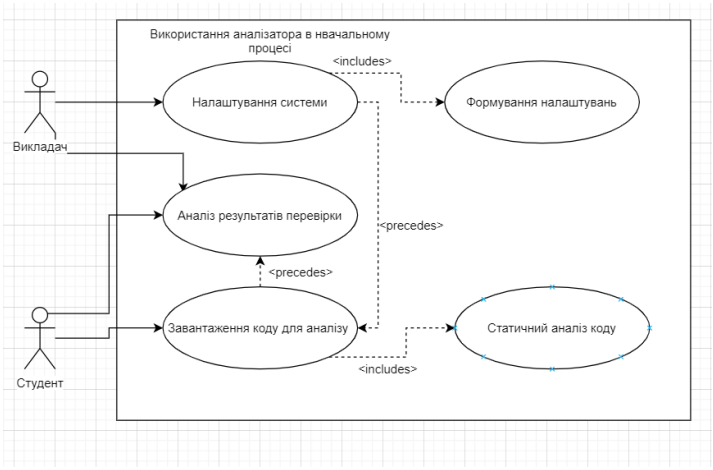


Рисунок 1.3 – Використання в навчальному процесі

Як бачимо, використання в навчальному процесі передбачене для 2 основних дійових осіб.

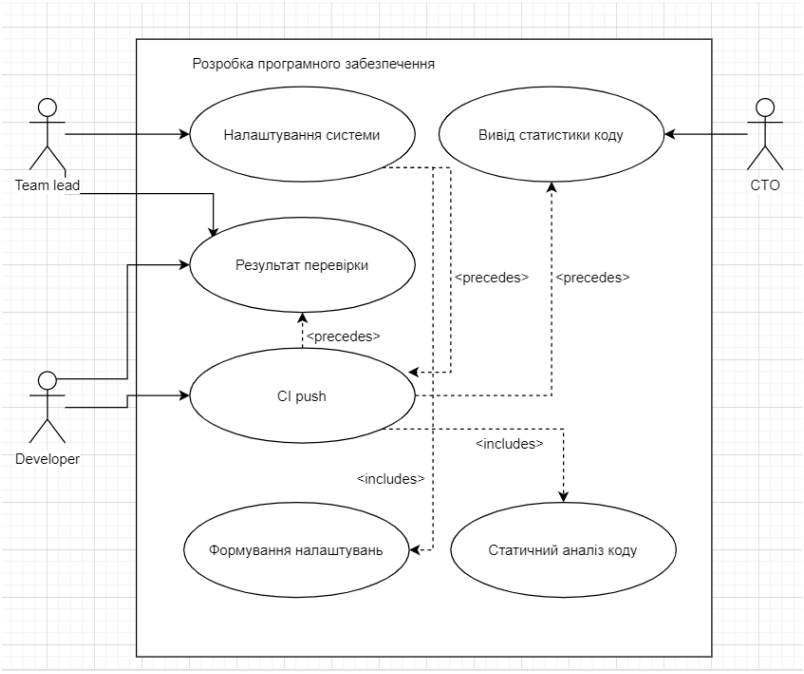


Рисунок 1.4 – Використання при розробці програмного забезпечення

При розробці програмного забезпечення склад дійових осіб дещо розширився за рахунок керівника, якому додатково доступна статистика зміни якості коду. Слід зазначити, що вказані варіанти використання застосовують деякі спільні варіанти використання для різних моделей використання.

1.4 Постановка задач дослідження

Призначенням розробки системи є створення програмного та математичного комплексу для аналізу та визначення характерних рис в коді програмного забезпечення.

Метою розробки є покращення процесу аналізу коду та виявлення небажаних вад, контролю якості, аналізу динаміки зміни коду в навчальному процесі чи на підприємстві, що дозволяє виконувати статичний аналіз коду, відтворювати результат аналізу в читабельній формі.

Для досягнення поставленої мети потрібно виконати такі задачі:

– дослідити методи статичного обробки та аналізу коду;

– виконати огляд існуючих інструментів, що виконують перевірку коду;

– вибрати алгоритми, що можуть стати основою для покращення;

– розробити алгоритм, що забезпечує збільшення інформації про код в процесі запуску;

– розробити програмну реалізацію алгоритму;

– провести порівняння якості отриманих результатів з існуючими відкритими аналогами;

– провести аналіз отриманих результатів.

Висновок до розділу

В даному розділі було проаналізовано предметну область, виділено основні аспекти роботи подібних систем, знайдено та порівняно подібні продукти на ринку з метою пошуку найкращих конкуретних рис для створюваного програмно-математичного комплексу.

На етапі опису предметного середовища було зазначено всі основні визначення та засоби, сформульовано їх визначення, які використовуються в подальших розділах.

На етапі опису процесу діяльності було описано процеси в яких буде брати участь розроблюваний комплекс та які він зможе покращити, особливості застосування комплексу.

При описі варіантів використання було зазначено дійові особи та операції, які вони виконують в системі, формючи таким чином варіанти використання. Було визначено, що проєкт містить два основних варіанти використання, які відрізняються набором прав дійових осіб.

При опису функціональної моделі було зазначено сценарії та варіанти використання комплексу. При огляді наявних аналогів було перелічено список основних аналогів, їх переваги та недоліки. Були визначені ціль та призначення розробки.

2 ТЕОРЕТИЧНІ ОСНОВИ

Одним із основних понять побудови алгоритму є абстрактне синтаксичне дерево[2], само воно використовується для аналізу виразів. Також додаткову інформацію надає семантична модель. Вона необхідна для зв’язку програмних структур в одну логічну мережу.

Крім того, вхідним кодом для статичного аналізу є код на об’єктно-орієнтованій мові C#, а отже до суті долучаються такі елементи як поліморфізм та інкапсуляція (наслідування в даному випадку на має ваги). Поліморфізм є важливою складовою для побудови гнучких програм, проте зі сторони аналізу він додає проблем, оскільки необхідно враховувати всі можливі шляхи виконання коду, тобото шукати всі поліморфні імплементації. Слід зазначити, що однією з основ цієї роботи є аналіз statefull-об’єктів, а отже інкапсуляція виступає центральним елементом для верифікації в алгоритмі.

Крім того одним із основних рушіїв алгоритму є алгоритм Хіндлі-Мілнера[3], що використовується в функціональних мовах програмування для автоматичного виводу типів. Зупинимось детально на кожному пункті.

2.1 Абстрактне синтаксичне дерево

AST (Абстрактне дерево синтаксису) – це графічне представлення вихідного коду, що в основному використовується компіляторами для читання коду та генерації цільових двійкових файлів.

Оброблення вихідного коду на AST є дуже поширеним шаблоном при статичному аналізу систем. Типовий робочий процес базується на створенні синтаксичного аналізатора[4], який перетворює необроблені дані у деревовидний формат, який потім може використовуватися механізмом верифікації.

В основному AST використовується під час семантичного аналізу, де компілятор перевіряє правильність використання елементів програми та мови.

Компілятор також генерує таблиці символів на основі AST під час семантичного аналізу. Повна обробка дерева дозволяє перевірити правильність програми.

Після перевірки правильності AST служить базою для генерації коду. AST часто використовується для генерації проміжного подання[5], яке іноді називають проміжною мовою, для генерації коду. На рисунку 2.1 зображено приклад структури дерева на псевдомові.

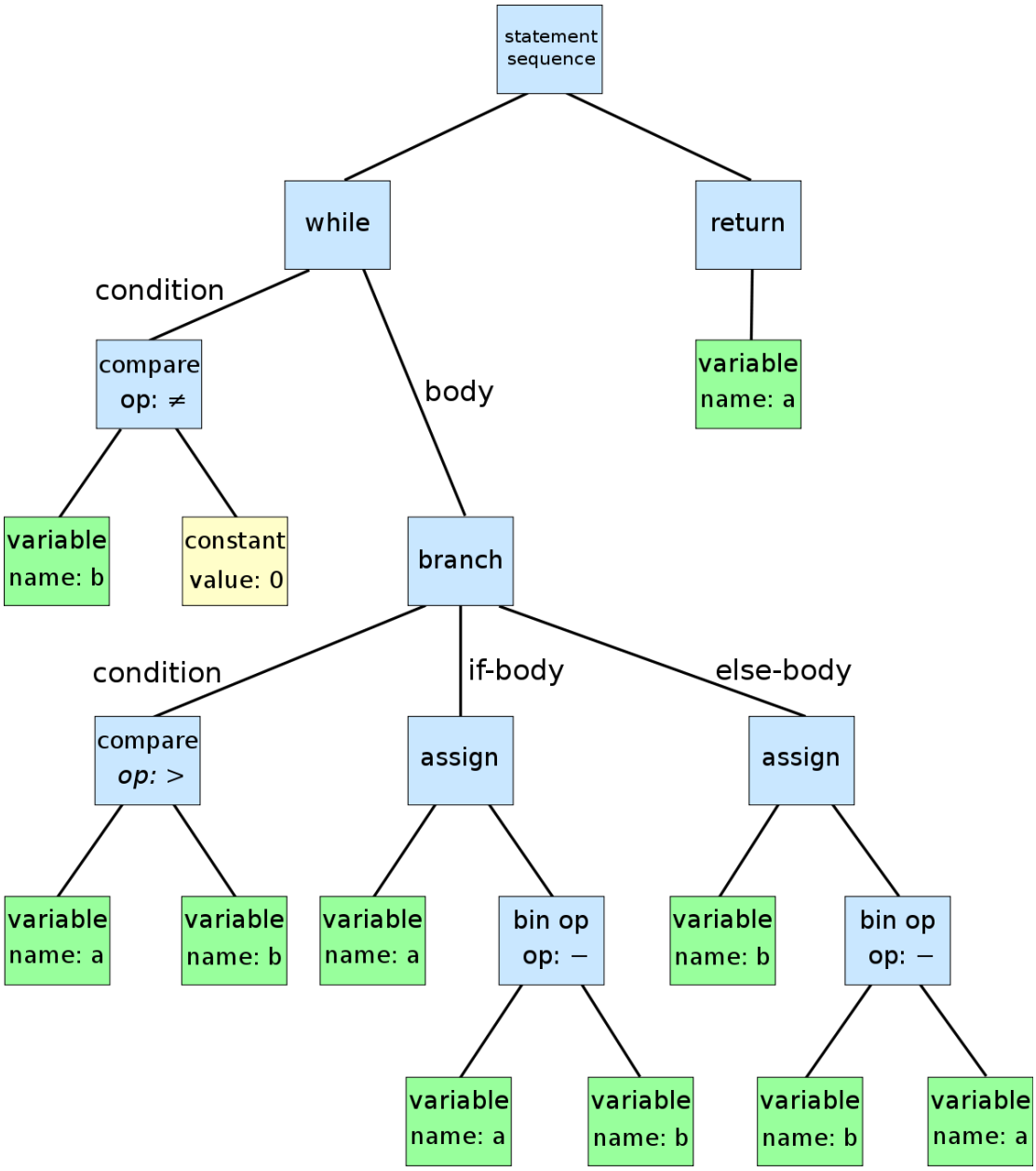


Рисунок 2.1 – Приклад AST дерева псевдомови

Крім того, одним із основних додаткових етапів перевірки правильності може бути верифікації коректності програмного коду.

2.2 Семантична модель коду

Семантика – це область, пов’язана з ретельним математичним вивченням значення мов програмування. Вона робить це, оцінюючи значення синтаксично допустимих рядків, визначених конкретною мовою програмування, показуючи обчислення, задіяне. У такому випадку, коли обчислення здійснюватиметься з синтаксично невірних рядків, результатом буде необчислення. Семантика описує процеси, яких дотримується комп’ютер під час виконання програми цією мовою. Це можна показати, описуючи взаємозв’язок між входом і виходом програми, або поясненням того, як програма буде виконуватися на певній платформі, отже, створюючи модель обчислень.

Для прикладу зобразимо семантичну модель коду на C# на рисунку 2.2.

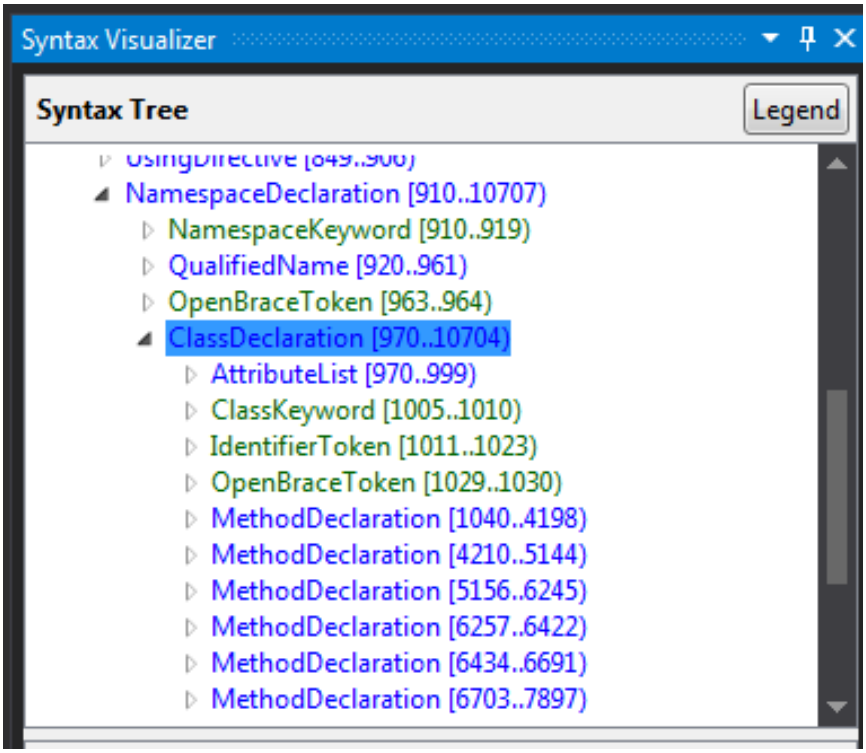


Рисунок 2.2 – Семантична структура C# коду

Як можна побачити на рисунку, семантична структура коду має ієрархічний вигляд та кожен елемент моделі містить дочірні елементи та токени.

2.3 Інкапсуляція

Інкапсуляція є однією з основ ООП[6]. Це стосується об’єднання даних із методами, що діють на ці дані. Інкапсуляція може використовується, щоб приховати значення або стан об’єкта структурованих даних всередині класу або іншого модулю, запобігаючи прямому доступу несанкціонованих сторін до них. Загальнодоступні методи, як правило, надаються в класі (так звані геттери та сеттери) для доступу до значень, а інші клієнтські класи викликають ці методи для отримання та модифікації значень в об’єкті. На рисунку 2.3 зображено схематично структуру побудови компонентів за допомогою інкапсуляції.

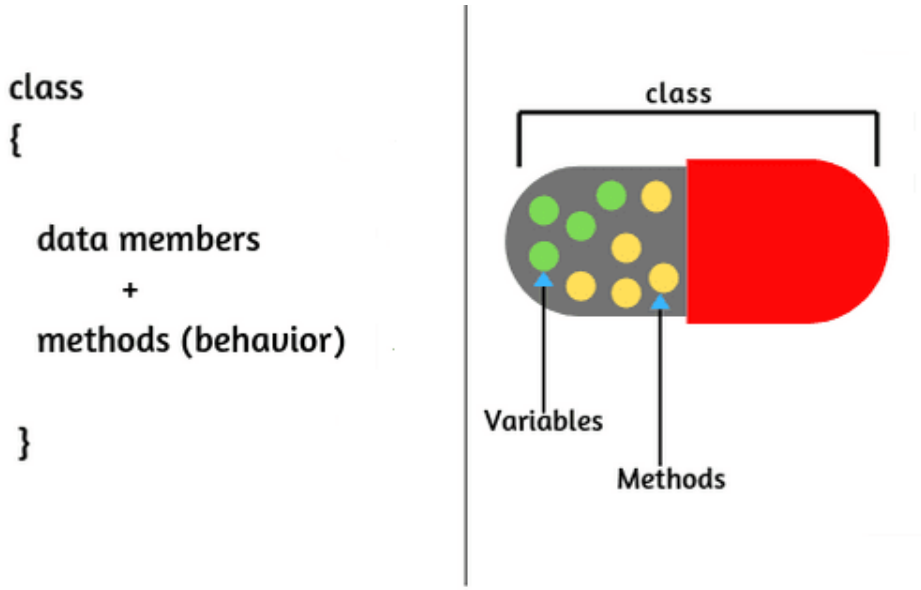


Рисунок 2.3 – Структура інкапсуляції

Відповідно до рисунку, інкапсуляція визначає методи (methods) та змінні (variables) як єдине ціле.

2.4 Поліморфізм

Поліморфізм – це надання єдиного інтерфейсу об’єктам різних типів або використання одного символу для представлення декількох різних типів[8]. Однією з основних властивостей поліморфізму зі сторони верифікації програмного забезпечення є те, що він вносить індірекцію в поток управління програма, а отже і ускладню аналіз, оскільки за одною частиною коду можуть бути різні імплеметації, що призводять до різних варіантів розпутування системи. Поліморфізм також має різні види, схематично зобразимо їх на рисунку 2.4.

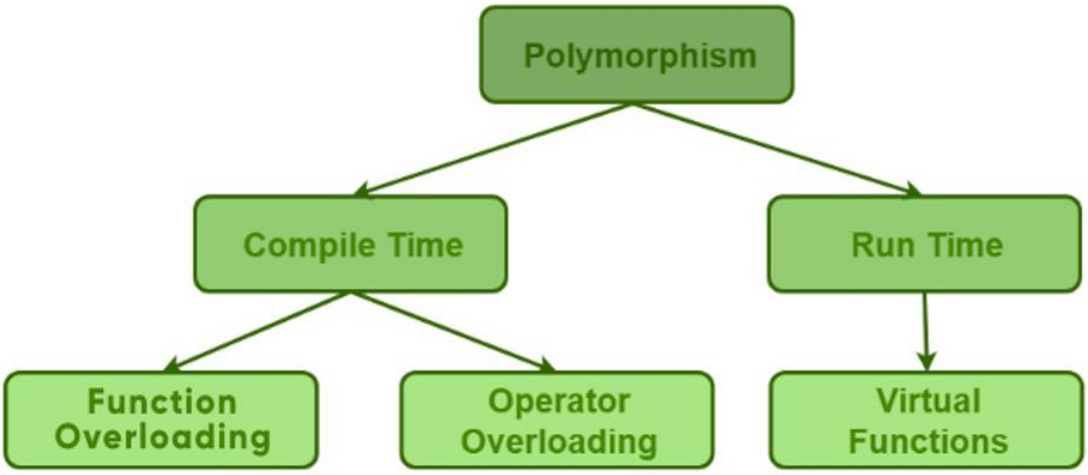


Рисунок 2.4 – Види поліморфізму

Як бачимо, поліморфізм має досить розгалужену структуру видів. Особливу увагу в даній роботі виділяється поліморфізму часу виконання(runtime), а саме побудованому на використанні віртуальних функцій.

2.5 Алгоритм Хіндлі-Мілнера для виводу типів

Алгоритм Хіндлі-Мілнера - механізм виведення типів виразів, що реалізовується в мовах програмування, заснованих на системі типів Хіндлі - Мілнера, таких як ML (перша мова цього сімейства), Standard ML, OCaml, Haskell, F #, Fortress і Boo. Мова Nemerle використовує цей алгоритм з рядом деяких необхідних змін.

Механізм виведення типів заснований на можливості автоматично повністю або частково виводити тип виразу, отриманого за допомогою обчислення деякого виразу. Так як цей процес систематично проводиться під час трансляції програми, транслятор часто може вивести тип змінної або функції без явної вказівки типів цих об'єктів. У багатьох випадках можна опускати явні декларації типів - це можна робити для досить простих об'єктів, або для мов з простим синтаксисом. Наприклад, в мові Haskell реалізований досить потужний механізм виведення типів, тому вказівка типів функцій в цій мові програмування не потрібно. Програміст може вказати тип функції явно для того, щоб обмежити її використання лише для конкретних типів даних, або для більш структурованого оформлення вихідного коду[7].

Система типів визначається в моделі Хіндли - Мілнера наступним чином:

- примітивні типи v є типами виразів;
- параметричні змінні типів α є типами виразів;
- якщо o_1 і o_2 - типи виразів, то тип o_1 -> o_2 є типом виразів;
- символ ⊥ є типом виразів.

Вирази, в яких типи обраховуються, визначаються в рамках системи досить стандартним способом:

- константи є виразами;
- змінні також є виразами;
- якщо e_1 і e_2- вирази, то (e_1, e_2) – вираз;
- якщо v - змінна, а e - вираз, то $\lambda v.e$ - вираз.

Розглянемо спосіб рішення системи рівнянь для виводу типів в системі Хіндлі-Мілнера. По-перше, потрібно вивести декілька обмежень, пов'язаних з цим. Почати можна з привласнення кожної змінної, або ж константи (x і y) нового типу (тобто «неіснуючого»). Якби було записано bar з цими типами, то вийшло б що-небудь на зразок *def bar(x: X, y: Y) = foo(x) + y.*

Не важлива назва типів, головне, що вони не мають нічного спільного з «реальними» типами. Вони потрібні при аналізі коду для того, щоб алгоритм, який це

робить, міг точно посилатися на ще не відомий тип для кожного значення. Без чого неможливо створити безліч обмежень.

Далі, алгоритм занурюється в тіло функції в пошуку операцій, які накладають деякі обмеження на тип. Першою операцією є виклик методу foo. Відомо, що тип foo це String => Int і це дозволяє нам записати обмеження $X \mapsto String$.

Наступною операцією буде +, пов'язаної зі значенням y. Відомо, що *foo (x)* це вислів типу *Int* (з типу що повертається foo), як і те, що + визначений як метод класу *Int* з типом *Int => Int*. Що дозволяє додати до наших обмежень ще одне: $Y \mapsto Int$.

Останній крок відновлення типів - це уніфікація всіх цих обмежень для того, щоб отримати реальні типи, які підставляються замість змінних X і Y. Уніфікація, буквально, - це процес проходу через всі обмеження в спробі знайти єдиний тип, який задовольняє їм усім.

У цьому прикладі уніфікація безлічі обмежень тривіальна. Було всього лише одне обмеження для кожного значення (x і y) і обидва вони відображаються в реальні типи. Єдине, що було потрібно це підставити "String" замість "X" і "Int" замість "Y". На рисунку 2.5 зображено приклад парсингу за допомогою цього алгоритму невеликої ділянки коду.

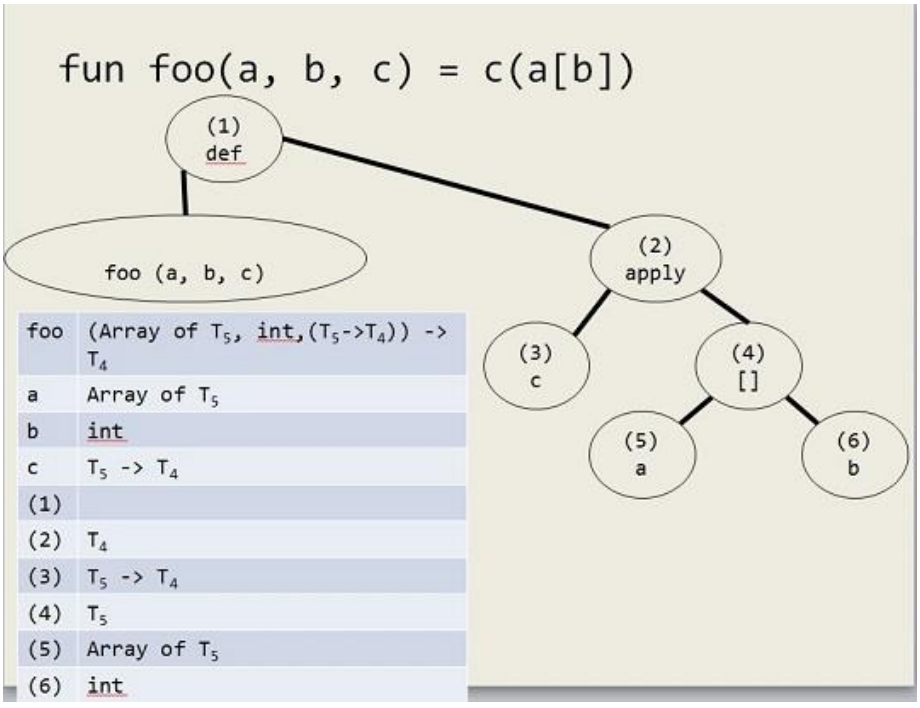


Рисунок 2.5 – Приклад розбору за Хіндлі-Мілнером

Як бачимо, принцип алгоритму Хіндлі-Мілнера має ієрархічну структуру перевірки типів. Слід зазначити, що розглянутий алгоритм, вирішує дещо іншу задачу, а саме вивід типів на основі коду, проте деякі його механізми можуть допомогти у вирішенні задачі статичного аналізу коду, наприклад алгоритм виводу типів на основі накладання обмежень на значення змінної.

Висновок до розділу

В розділі наведено основні теоретичні складові для алгоритму статичного аналізу коду. Слід зазначити, що розглянута модель даних є універсальною для систем перевірки коду, а також компіляторів, що обумовлює її стандартизацію, а отже її можна використовувати для цілого класу алгоритмів обробки формальних мов. Задекларовані структури даних будуть використовуватись для синтезу модифікованого алгоритму. Також буде взято деякі механізми з зазначених підходів до типізації для отримання покращеного алгоритму статичного аналізу коду.

3 АНАЛІЗ МЕТОДІВ СТАТИЧНОГО АНАЛІЗУ КОДУ

Серед алгоритмів, що можуть аналізувати потік управління найбільш поширеними є data-flow аналіз[9], taint аналіз та структурний аналіз в мові Typescript. Розглянемо існуючі підходи в розробці програмного забезпечення для аналізу їх характерних рис, переваг та недоліків при статичному аналізі коду та оберемо базовий алгоритм для модифікації, що забезпечує необхідні функціональні можливості.

3.1 Data-flow аналіз

Основою алгоритму є трьохетапна обробка даних, а саме: парсинг, побудова зв’язків між елементами програми та аналіз зв’язків на використовувані паттерни.

Розглянемо детально третій етап, в якості даних він використовує 2 графи – граф вхідного коду та граф бібліотеки шаблонів.

Граф вхідного коду включає в себе: оголошення класів, оголошення інтерфейсів, публічні методи, властивості та їх типи[10]. Використовуючи тіло класів та інтерфейсів можна визначити посилання на інші. Таким чином в структурі графа будуть вершини, що представляють класи та ребра що представляють залежності класів.

Кожен конкретний тип аналізу потоку даних має свою особливу функцію передачі та операції приєднання. Деякі проблеми аналізу потоку даних вимагають аналізу зворотного потоку. Він працює так же, за винятком того, що функція передачі застосовується до стану виходу, що дає стан входу, а операція приєднання працює над станами входу наступників, щоб отримати стан виходу.

Найбільш поширеним способом вирішення рівнянь потоку даних є використання ітераційного алгоритму.Оскільки префіксні дерева побудовані таким чином, що усі ключі зі спільним префіксом мають спільні вузли які відповідають цьому префіксу, то це дає можливість здійснювати пошук не маючи повного ключа,

а тільки його префікс, а результатом буде набір можливих ключів які зберігаються у дереві.

Починається з апроксимації стану кожного блоку. Потім вихідні стани обчислюються шляхом застосування передавальних функцій у внутрішніх станах. З них внутрішні штати оновлюються, застосовуючи операції об’єднання. Два останні кроки повторюються, поки ми не досягнемо так званої точки фіксування : ситуації, в якій внутрішні (і, як наслідок, зовнішні стани) не змінюються.

Основним алгоритмом вирішення рівнянь потоку даних є ітераційний круговий ітераційний алгоритм, що зображений на рисунку 3.1.

```
для і ← 1 до N
    ініціалізувати вузол і
while ( набори все ще змінюються )
    для і ← 1 до N
        перерахувати набори у вузлі і
```

Рисунок 3.1 – Обхід вузлів дерева в аналізі потоку даних

Щоб бути придатним для використання, ітераційний підхід повинен фактично досягти точки фіксації. Це може бути гарантоване накладенням обмежень на комбінацію області значень станів, функцій передачі та операції об'єднання.

Форвадний аналіз – це аналіз визначень, що обчислює для кожної точки програми набір визначень, які потенційно можуть досягти цієї точки програми[11]. Для прикладу код на Рис. 1, на якому визначальне значення змінної a у рядку 10 - це набір призначень a = 5 у рядку 5 та a = 3 у рядку 7.

Аналіз назад – аналіз змінних в реальному часі, що обчислює для кожної точки програми змінні, які потенційно можуть бути прочитані після цього перед наступним оновленням запису. Результат, як правило, використовується шляхом усунення мертвого коду для видалення операторів, які присвоюються змінній, значення якої потім не використовується.

Внутрішній стан блоку - це набір змінних, які працюють на початку його. Спочатку він містить усі змінні в передачі та обчислюватимуться фактичні значення, що містяться. Функція передачі оператора застосовується вбивством змінних, записаних у цьому блоці (вилучіть їх із набору змінних, що працюють). Зовнішній стан блоку - це набір змінних, які працюють в кінці блоку, і обчислюється об'єднанням внутрішніх станів наступників блоку. Розглянемо початковий код на рисунку 3.2.

```
1   let b = 4;
2   let a, x, d, c = 5;
3
4   b1: a = 3;
5       b = 5;
6       d = 4;
7       x = 100;
8       if(a > b) {
9   b2:     c = a + b;
10          d = 2;
11  b3: }
12      c = 4;
13      return b * d + c;
14
```

Рисунок 3.2 - Приклад коду для зворотнього аналізу

Функція передачі оператора застосовується вбивством змінних, записаних у цьому блоці (вилучіть їх із набору змінних, що працюють)[12]. Зовнішній стан блоку - це набір змінних, які працюють в кінці блоку, і обчислюється об'єднанням внутрішніх станів наступників блоку. Розглянемо початковий код на рисунку 3.2.

Проаналізуємо цей приклад. Вхідний стан b3 містить лише b і d , оскільки c написано. Зовнішній стан b1 - це об’єднання внутрішніх станів b2 і b3. Визначення c з b2 можна вилучити, оскільки c не діє безпосередньо після оператора.

Вирішення рівнянь потоку даних починається з ініціалізації всіх вхідних та вихідних станів до порожнього набору. Робочий список ініціалізується шляхом вставки точки виходу (b3) у робочий список (типово для зворотного потоку). Його

обчислюваний внутрішній стан відрізняється від попереднього, тому його попередники b1 та b2 вставляються, і процес продовжується. Зверніть увагу, що b1 було внесено до списку перед b2, що примусило обробляти b1 двічі (b1 було повторно введено як попередник b2). Вставка b2 перед b1 могла б дозволити більш раннє завершення.

Ініціалізація порожнім набором є оптимістичною ініціалізацією: всі змінні починаються як мертві. Слід зазначити, що зовнішні стани не можуть зменшуватися від однієї ітерації до наступної, хоча зовнішній стан може бути меншим, ніж внутрішній. Це видно з того факту, що після першої ітерації зовнішній стан може змінитися лише шляхом зміни внутрішнього стану. Оскільки in-state починається як порожній набір, він може зростати лише в подальших ітераціях.

Відобразимо наглядно приклад для цього блоку на рисунку 3.3.

```
1
2   // y: {}
3   b1: a = 3;
4       b = 5;
5       d = 4;
6       x = 100; // x ніколи не використовується пізніше, отже, не в наборі {a, b, d}
7       if (a > b) {
8   // out: {a, b, d} // об'єднання всіх (не) наступників b1 => b2: {a, b} та b3: {b, d}
9
10  // y: {a, b}
11  b2: c = a + b;
12      d = 2;
13  // вихід: {b, d}
14
15  // y: {b, d}
16  b3: }
17      c = 4;
18      return b * d + c;
19  // вихід: {}
```

Рисунок 3.3 - Приклад станів системи при проведенні «аналізу назад»

Як можна бачити з наведеного рисунку, «аналіз назад» змінює стан системи в зворотньому порядку, а отже і обчисляє декларації відповідно до операції повернення зі функції.

3.2 Аналіз контрольної залежності

Дерева домінування опускають важливий аспект інформації: умови. Вихідний код складається з логічних конструкцій, таких як блоки if-then-else та for loop. Виконання заяви, як правило, залежить від результатів прийняття рішень протягом усієї програми. Цей умовний зв’язок між висловлюваннями не фіксується лише домінуванням[13].

Аналіз контрольної залежності ґрунтується на аналізі домінування та аналізі потоку даних, щоб охопити умовні зв’язки між твердженнями. Оператор залежить від висловлювання, якщо виконання визначається результатом, але не домінує. Іншими словами, виконання мусить залежати виключно від оцінки предиката; якщо виконується незалежно від результату, він не залежить від контролю.

Ці взаємозв’язки контрольної залежності можна знову переглянути в графічному форматі, що називається графіком контрольної залежності (CDG). CDG для прикладу Bounce.bas показано на рисунку 3.3. За замовчуванням будь-які оператори верхнього рівня, які не містяться в умовному блоці, залежать від керування вхідним вузлом до CFG. Вкладені гілки позначають вкладені умовні блоки.

Схематично цей різновид аналізу представлений на рисунку 3.4.

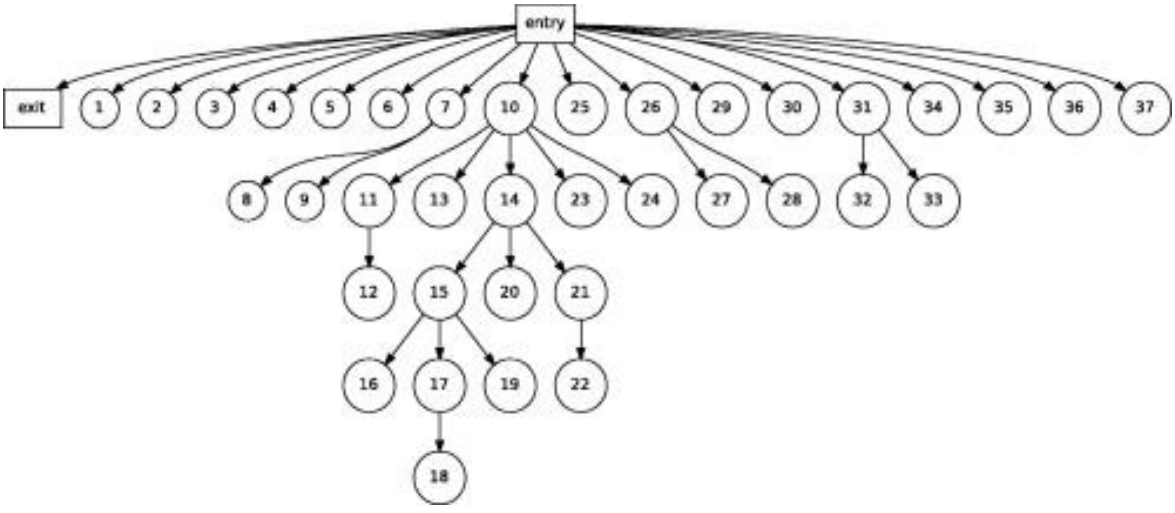


Рисунок 3.4 – Графік контрольної залежності

Як бачимо система аналізу має досить розгалужений вигляд у представленому графі, оскільки виконує комбінування всіх умов в коді.

3.3 Taint аналіз

Окремо варто зупинитися на одній із завдань аналізу потоку даних - taint-аналізі. Taint-аналіз дозволяє поширити по програмі прапори[14]. Дане завдання є ключовою для інформаційної безпеки, так як саме за допомогою неї виявляються уразливості, пов'язані з впровадженням даних (впровадження в SQL, міжсайтовий скриптинг, відкриті перенаправлення, підробка файлового шляху і так далі), а також з витоком конфіденційних даних (записи паролей в журналі подій, небезпечна передача даних).

Розглянемо наступний приклад: нехай потрібно відстежити *n* прапорів *f1, f2…fn*. Множиною інформації тут буде множина підмножин *{f1…fn}* так як для кожної змінної в кожній точці програми необхідно визначити її прапори.

Далі потрібно визначити функції потоку. В даному випадку функції потоку можуть визначатися такими міркуваннями:

- задано множину правил, в яких визначені конструкції, що призводять до появи або зміни набору прапорів;

- операція присвоювання перекидає прапори з правої частини в ліву;

- будь-яка невідома для множин правил операція об'єднує прапори з усіх операндів і підсумкове безліч прапорів додається до результатів операції.

Отже, потрібно визначити правила злиття інформації в точках з'єднання дуг CFG. Злиття визначається за правилом об'єднання, тобто якщо з різних базових блоків прийшли різні набори прапорів для однієї змінної, то при злитті вони об'єднуються. У тому числі звідси з'являються помилкові спрацьовування: алгоритм не гарантує, що шлях в CFG, на якому з'явився прапор, може бути виконаний.

Наприклад, необхідно виявляти вразливості типу «ін'єкції в SQL» (SQL Injection)[15]. Така вразливість виникає, коли неперевірені дані від користувача потрапляють в методи роботи з базою даних. Необхідно визначити, що дані надійшли

від користувача, і додати таким даними прапор taint. Зазвичай в базі правил аналізатора задаються правила постановки прапора taint. Наприклад, поставити прапор що повертається значенням методу getParameter () класу Request.

3.4 Структурний аналіз в мові Typescript

Мова програмування Typescript містить декілька цікавих, з точки зору типізації, речей, зокрема, вона здатна уточнювати тип змінної після проходження перевірки в умовних блоках, тернарних операторах, таким чином, маючи більш уточнений вивід типу можна його використовувати в тих операціях, що приймають часткове значення. Проілюструємо це прикладом, що наведений на рисунку 3.5.

```
126
127  let name: string = "Yurii" // тип string
128
129  function sayHello(to: "Yurii" | "Ivan") {
130      console.log(`Hello ${to}`)
131  }
132       let name: string
133  if (name === "Yurii") {
134      sayHello(name)
135  }
```

Рисунок 3.5 – Приклад Typescript аналізу

Тобто компілятор статично визначає, що в цьому блоці змінна буде мати більш конкретне значення. Також тут наявні guard-функції, які виконують задачу перевірки відношення змінної до певного типу. Структура типів, яка дозволяє обробку в такий спосіб, має деяку специфіку, зокрема типи any(тип, що є сумісним з будь-яким іншим типов), unknown(тип, що базовим для будь-якого іношого типу) та never(тип, який є підтипом будь-якого іншого типу). Детально ієрархію типів зобразимо на рисунку 3.6.

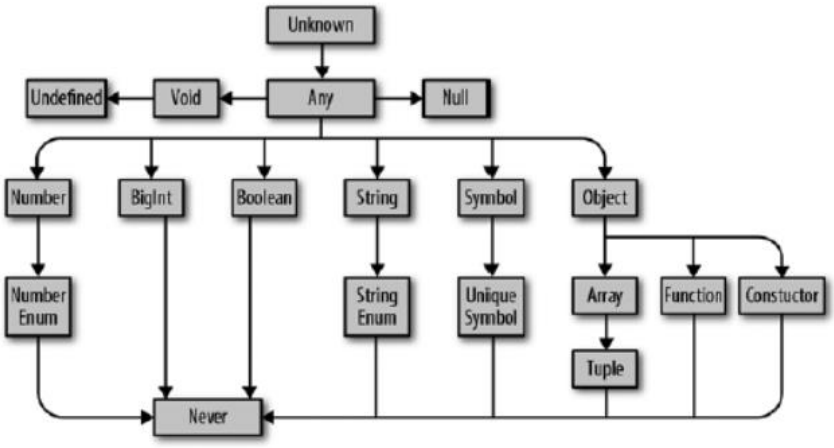


Рисунок 3.6 – Структура типів Typescript

Згідно структурі типів в мові, існують як верхні обмеження так і нижні, які представляються ключовим словом never.

3.5 Аналіз методів статичного аналізу коду

Складемо порівняльну таблицю методів статичного аналізу коду (табл. 3.1).

Таблиця 3.1 – Порівняння методів

Назва методу	**Переваги**	**Недоліки**
Data-flow аналіз	Отримання можливих даних в будь-якій точці коду, можливість виводу констант для оптимізацій коду	Перевірка вхідних ззвоні даних
Taint-аналіз	Аналіз вхідних ззовні даних	Замало даних, лише часткова перевірка коду з метою пошуку вразливостей SQL-ін’єкцій в коді

Продовження таблиці 3.1

Назва методу	**Переваги**	**Недоліки**
Структурний аналіз в мові Typescript	Працює статично, під час розробки коду, є стркутурно гнучким	Дещо обмежений, в тому числі і в системі типів, оброці даних, що надійшли ззовні
Аналіз контрольної залежності	Аналізує всі умовні конструкції	Має обмежений в рамках типів аналіз.

Отже, на основі порівняльної таблиці можна зробити наступні висновки.

Алгоритм аналізу data-flow створює структуру даних для каждої інструкції намагаючись мати визначити значення[14], яке буде в процесі виконання. Одним із можливих застосувань цього – компіляторна оптимізація constant propagation. Проте цей підхід не може буде використаний у випадку, коли перевіряється код, що обробляє дані, що найдійшли із-зовні, оскільки він зав’язані тільки на конкретні можливі значення. Також даний підхід гарно підходить в основному для процедурного програмування[16], тобто, при перевірки за допомогою нього об’єктів зі станами ми матимемо проблему з правильним визначенням значення в коді використання. Звичайно, є способи обійти цю проблему, як-от міжпроцедурний аналіз з комбінацією вхідних даних, проте він значно збільшує час виконання аналізу за рахунок цієї комбінації.

Алгоритм taint аналізу має наступну цікаву відмінність, яка полягає в тому, що він здатен оперувати прапорцями наявності зовнішнього вводу. Це є протилежним аспектом data-flow аналізу оскільки він використовує мітку не про фактичне значення в змінній, а про її стан в системі, саме це буде використано в модифікованому алгоритмі, поєднання значень, об’єднань та типу any<T>, що характеризує будь-яке можливе значення (тобто прийшло з зовні та не є перевіреним).

Алгоритм виводу типу в Typescript є одним із основних ідейних джерел цього алгоритму, оскільки він оперує типами відносно змінних на основі структури

операцій, проте він не враховує механізми інкапсуляції та не може вивести тип змінюваного об’єкта, що і є його основним недоліком, що не дає змогу використовувати його в якості опорного методу аналізу.

Висновки до розділу

В даному розділі було проведено аналіз, та розглянуто існуючі методи проведення статичного аналізу, що включає перевірку data-flow на деякі характеристики. В якості опорного методу статичного аналізу коду обрано алгоритм data-flow, так як він містить основну необхідну логіку для виконання статичного аналізу, але варто включити частину механіхму інших методів, зокрема підхід структурованого виводу типу в мові Typescript, з якого можна взяти ідею обчислення типу в середині конструкції на основі умовних конструкцій, та taint-аналізу, у якого можна взяти складову додавання прапорців до інструкцій та змінювати їх стан в процесі виконання коду. Загальні механізми аналізу мають дещо спільні риси – прописаний стан для кожної інструкції, єдине, що відрізняється це структура стану, вона визначає те, що буде перевірятись та які результати буде отримано.

4 ОПИС ПРОГРАМНОГО ПРОДУКТУ

4.1 Архітектура програмного забезпечення

Зобразимо діаграму архітектури програмного забезпечення на риснку 4.1 та проведемо детальний розбір її компонентів в наступних розділах.

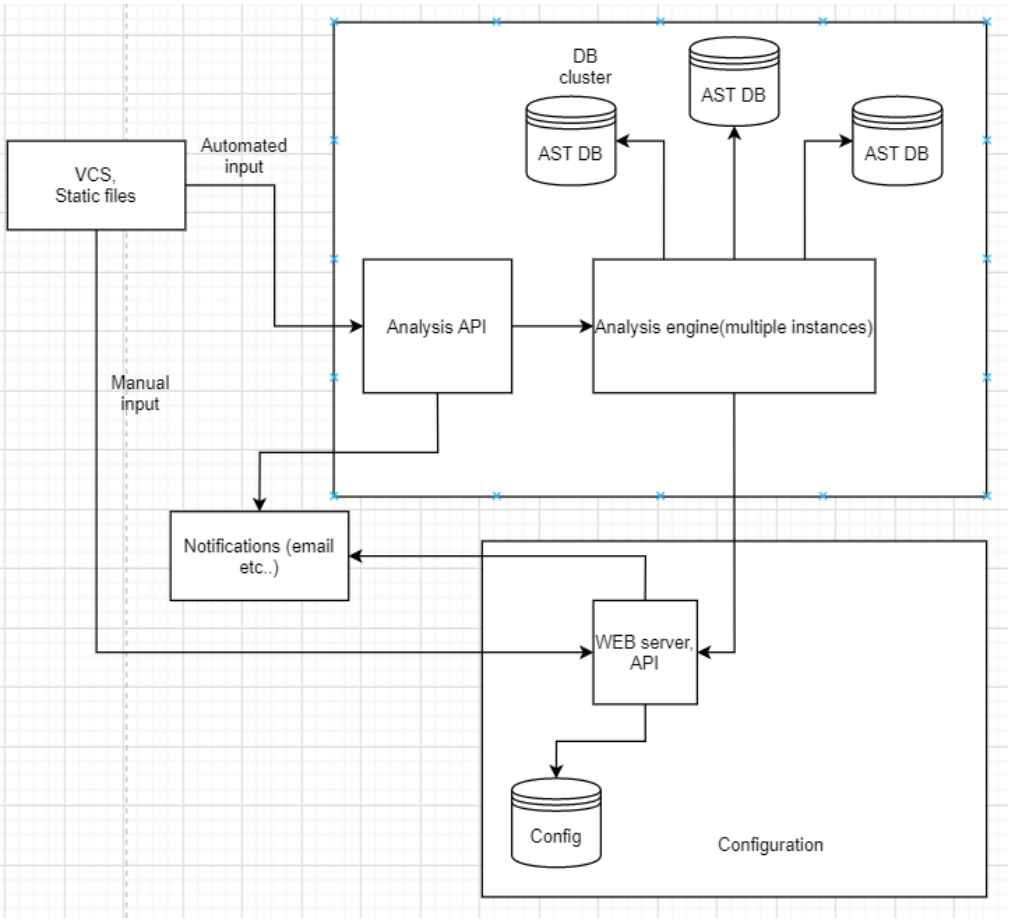


Рисунок 4.1 – Архітектура програмного забезпечення

Система, зображена на рисунку, складається з незалежних компонентів, кожен з яких виконує окрему функціональну роль.

4.2 Засоби розробки

Засобами розробки виступають мови програмування C#, Typescript, F#. Функціонал сервісу можна поділити на такі складові:

- клієнтська частина;
- сервер для оборобки запитів;
- алгоритм статичного аналізу;
- база даних.

Клієнтська частина складається за React та Typescript. Вибір саме на цих інструментах, оскільки вони надають можливість швидко та зручно побудувати надійний користувацький інтерфейс. Найдійність полягає в типазації мови програмування.

Серверна частина складається з прийому API запитів, який написаний на платформі Node.js та мові Typescript, синтаксичного аналізу, який написаний на C#, оскільки бібліотека для аналізу C# коду написана на цій мові.Сам алгоритм написано на мові Typescript з метою зручної обробки структур даних, метапрограмування, інструменту для гнучкої типізації часу виконання. Додаткові сервіси, як-от нотифікації, написані на F#.

База даних в сервісі потрібна для налаштувань користувачів та проектів та збереження проміжних даних структури AST, тому була вибрана документно-орієнтована база даних MongoDB.

4.2.1 Roslyn

Для виділення AST та семантичної моделі з вихідного коду було вибрано бібліотеку Roslyn, яка має базовий набір, необхідний для побудови аналізаторів.

Roslyn - це платформа з відкритим кодом, розроблена Microsoft, яка включає компілятори та інструменти для аналізу та аналізу коду, написаного мовами програмування C # та Visual Basic. Roslyn використовується в середовищі розробки Microsoft Visual Studio 2015. Різні нововведення, такі як виправлення коду, можуть бути впроваджені лише за допомогою Roslyn.

За допомогою інструментів аналітики, наданих платформою Roslyn, ви можете повністю проаналізувати код, проаналізувавши всі підтримувані мовні структури. Далі не є єдиною версією статичного аналізатора та правил діагностики. Діагностика

може бути проведена за допомогою стандартного класу DiagnosticAnalyzer. Вбудована діагностика Roslyn використовує це рішення. Це дозволяє, наприклад, інтегрувати зі стандартним списком помилок Visual Studio, виділяти помилки в текстовому редакторі тощо, але якщо цей діагноз знаходиться у процесі devenv.exe, майте на увазі, що це буде 32. -bit, існують значні обмеження щодо обсягу використовуваної пам'яті. На рисунку 4.2 зображено стандартне вікно створення проекту для аналізу коду.

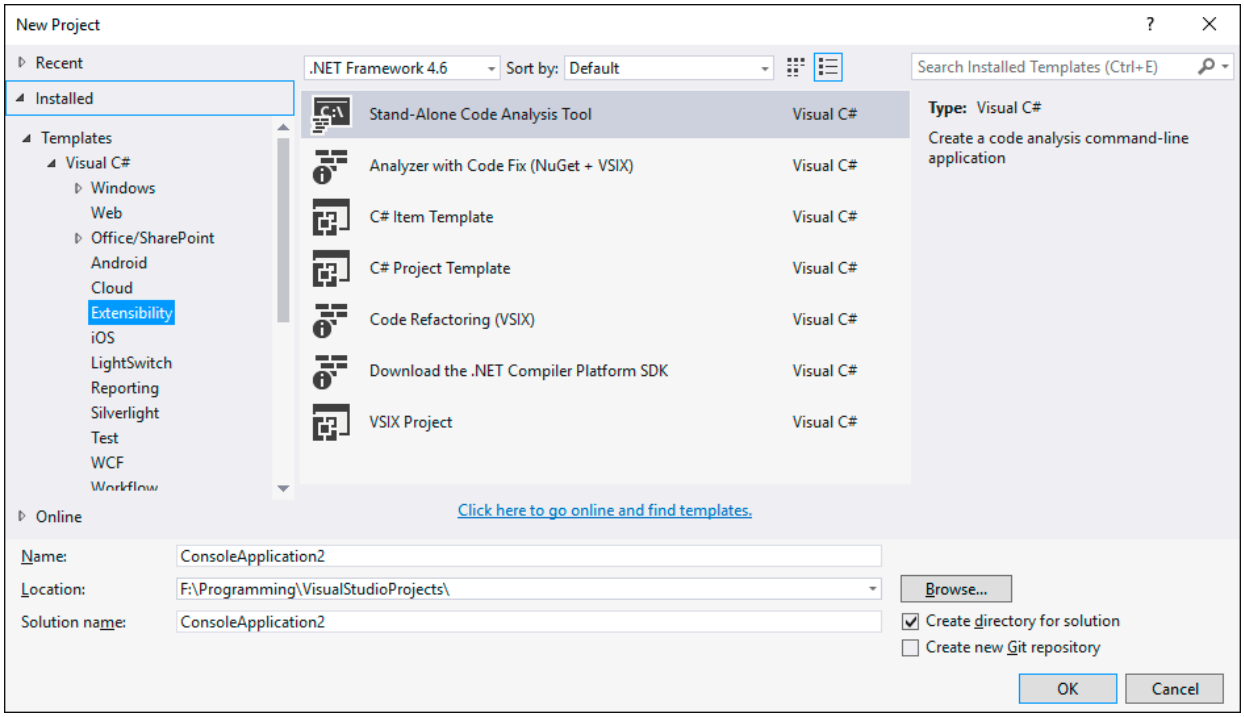


Рисунок 4.2 – Приклад інтерфейсу створення проєкту аналізу

У деяких випадках це дуже важливо і не дозволяє поглиблено аналізувати великі проекти. Крім того, у цьому випадку Roslyn залишає забудовнику мало контролю над проходженням дерев і робить процес незалежно паралельним.

4.2.2 MongoDB

В якості системи управління базами даних в наслідок відсуності небхідності в складних обрахунках та консистентності, а з іншого боку в потребі в збереженні нестурктурованих даних було вибрано MongoDB.

MongoDB - це база даних, орієнтована на документи для роботи, на декількох платформах. MongoDB класифікується як програма баз даних NoSQL і використовує подібну до JSON документацію з додатковими схемами. MongoDB, MongoDB Inc., розроблена. та ліцензується за Загальною ліцензією сервера (SSPL).

MongoDB підтримує пошук у полях, інтервалах та регулярних виразах. Запити можуть повертати певні поля в документах, а також можуть містити визначені користувачем функції. Запити також можна налаштувати на повернення випадкового вибору результатів певного розміру[15].

Індексація поля в документі MongoDB можна індексувати за допомогою первинного та вторинного індексів.

Реплікація MongoDB забезпечує високу доступність наборів реплік. Набір реплік складається з двох або більше копій даних. Кожен член набору реплік може служити первинною або вторинною реплікою в будь-який час. За замовчуванням усі написання та читання виконуються на первинній карті. Вторинні дублікати зберігають копію первинних даних із вбудованою реплікацією. Коли первинне відображення не вдається, набір відображень автоматично проходить процес відбору, щоб визначити, яке вторинне відображення має бути основним. Вторинні дані також можуть служити показаннями, але за замовчуванням ці дані в кінцевому підсумку узгоджуються.

Балансування навантаження MongoDB масштабується горизонтально з поділом. Користувач вибирає кнопку відстеження, яка визначає спосіб розподілу даних у колекції[17]. Дані поділяються на інтервали (на основі пакетного ключа) і охоплюють кілька груп. (Доріжка - це головна копія з однією або кількома копіями.) Або ключ доріжки можна хешувати відповідно до доріжки, щоб забезпечити рівномірний розподіл даних.

MongoDB можна запускати на декількох серверах, балансування навантаження або реплікацію даних, щоб забезпечити роботу системи у випадку несправності обладнання.

Зберігання файлів MongoDB може використовуватися як файлова система під назвою GridFS з функціями балансування навантаження та реплікацією даних на декількох комп'ютерах для зберігання файлів.

Ця функція, яка називається сітковою файловою системою, включена до драйверів MongoDB. MongoDB надає розробникам функції для роботи з файлами та вмістом. Доступ до GridFS можна отримати за допомогою інструменту монгофайлів або плагінів для Nginx та lighttpd. GridFS ділить файл на фрагменти або фрагменти та зберігає кожну частину як окремий документ.

MongoDB пропонує три способи агрегування: конвеєр вибору, функція обрізання карти та методи агрегування одноразового використання.

Може використовуватися для мінімізації карт, пакетної обробки та агрегування. Однак, згідно з документацією MongoDB, трубопровід вибору забезпечує найкращу продуктивність для більшості агрегатів.

Структура агрегування дозволяє користувачам отримувати результати за допомогою оператора SQL GROUP BY. Оператори агрегації можуть бути розміщені в конвеєрах, подібно до конвеєру Unix. Структура агрегування включає оператор пошуку $, який може поєднувати документи з декількох колекцій та статистичні оператори, такі як стандартне відхилення[18].

JavaScript можна використовувати в запитах, функціях агрегування (наприклад, MapReduce) і надсилати безпосередньо до бази даних для виконання.

Приватні колекції MongoDB підтримує колекції фіксованого розміру, які називаються обмеженими колекціями. Цей тип колекції підтримує порядок вставки та поводиться як кругова черга, коли досягається вказаний розмір.

MongoDB стверджує, що підтримує транзакції ACID з кількома документами, починаючи з версії 4.0 у червні 2018 року. Це твердження виявилося неправдивим, оскільки MongoDB порушило моментну ізоляцію..

Функціональна архітектура MongoDb зображена на рисунку 4.3 [19].

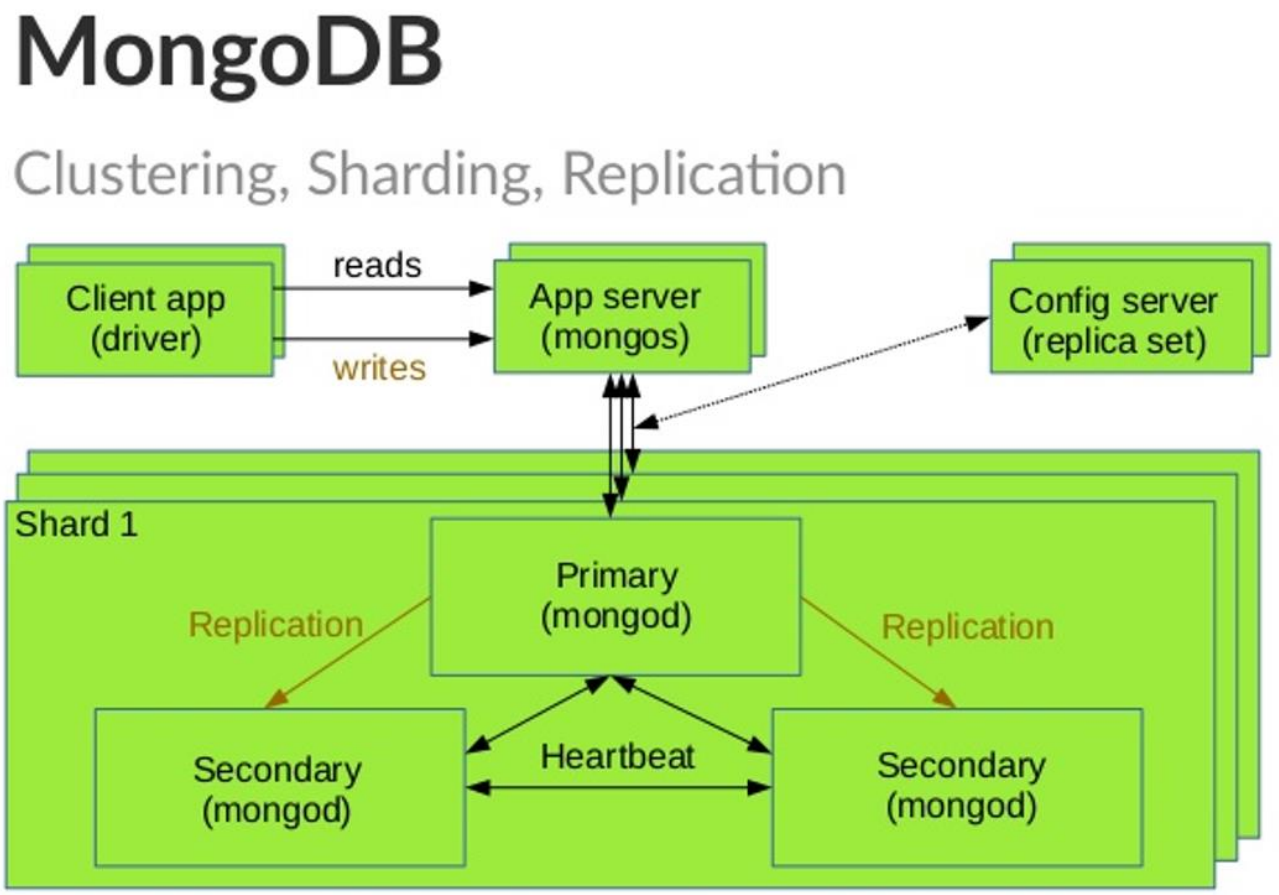


Рисунок 4.3 – Архітектура MongoDb

4.2.3 Fastify

В якості серверної частини проєкту було вибрано бібліотеку Fastify, оскільки вона має весь необхідний функціонал для цього, написана на мові Typescript.

Fastify – це веб-система, орієнтована на забезпечення найкращого досвіду для розробників з мінімальними накладними витратами та потужною архітектурою плагінів. Він був натхненний Hapi та Express, і, наскільки відомо, це один з найшвидших двигунів серверних застосунків у місті.

Ось основні особливості та принципи, на яких він базується:

– висока продуктивність, fastify – одна з найшвидших веб-систем у місті, враховуючи складність коду, можена обробляти до 30 000 запитів в секунду;

– розширюваний fastify можна повністю розширити за допомогою гачків, плагінів та декораторів;

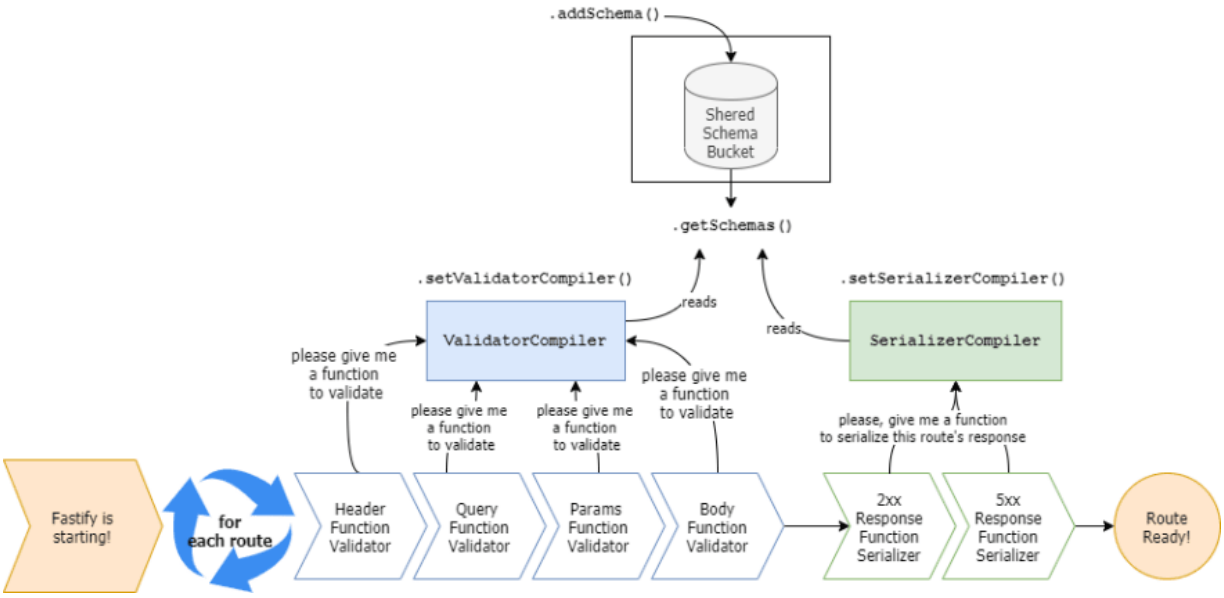
– розклад хоча не потрібний, рекомендується використовувати розклад JSON для перевірки маршрутів та серіалізації результатів. Fastify надає потужну функцію планування зсередини;

– вхід в журнали надзвичайно важливі, але дорогі; обраний найкращий алгоритм, щоб майже усунути ці витрати;

– придатний для розробників: система створена з високою виразністю та допомагає розробникам користуватися ними щодня, не жертвуючи продуктивністю та безпекою;

– можливості TypeScript: Докладається всі зусилля, щоб підтримувати файл декларації типу TypeScript для підтримки зростаючої спільноти TypeScript.

На рисунку 4.4 зображена архітектура побудови застосунку, використовуючи Fastify[20].



Рискноу 4.4 – Серверна архітектура на основі Fastify

Представлена архітектура є типовою для більшості проектів, побудованих з застосуванням Fastify.

4.3 Вимоги до технічного забезпечення

Для роботи програмного забезпечення необхідно:

– мати комп’ютер за ОС Linux або Windows (тестувалось лише на Windows);

– оперативну пам’ять – мінімум 4ГБ, цей показник може залежати від розміру проектів, найбільш оптимальним мінімальним показником для проєктів різних розмірів є 8 ГБ;

– необхідно мати CPU за тактовою частотою мімнімум 1 GHz.

4.4 Опис архітектури програмного забезпечення

Програмне забезпечення складається за класів та інтерфейсів.

4.4.1 Опис класів

Структура класів наведена в таблиці 4.1.

Таблиця 4.1 – Структура класів

Клас	**Опис**
ProjectController	Клас, що представляє з себе контроллер для обробки клієнтських запитів для обробки проєкту та є частиною тришарової архітектури.
SettingsController	Клас для обробки запитів налаштувань, містить метод для оновлення, додавання та повернення до стандартних налаштувань у системі сервісу
NotificationController	Клас для обробки запитів щодо налаштувань нотифікацій відносно процесу аналізу

Продовження таблиці 4.1

Клас	**Опис**
AbstractSyntaxTreeParser	Клас, що виконує обробку та парсинг синтаксичного дерева коду, є основним класом, що посилається на бібліотеку Roslyn
ProjectRepository	Клас, що є складовою тришарової архітектури та необхідний для покриття всіх запитів да бази даних відносоно моделі проектів, включає в себе функціонал для оброблення проміжних абстрактних синтаксичних дерев.
AlgorithmWalker	Центральний клас реалізації алгоритму розробору дерева та простановки типізованих міток, виконує поліморфний статичний аналіз та формує результуючий об’єкт проекту
ProjectService	Сервіс, який містить в собі основну логіку для виклику парсингу, нотифікацій та виклику алгоритму аналізу коду, також виконує планування розподіленого аналізу по обчислювальним нодам
NotificationService	Сервіс для управління нотифікаціями в проєкті
ProjectLoader	Клас, що здійснює завантаження проекту в структури сервісу

Продовження таблиці 4.1

Клас	**Опис**
LoginPage	Клієнтський клас, що містить в собі логіку та користувацький інтерфейс для керування аккаунтом користувача
Utils	Утилітарний клас, що містить універсальні функції для роботи з базовими та специфічними стрктурами даних
ProjectPage	Клієнтський клас, що містить в собі логіку та користувацький інтерфейс для керування завантажуваними для перевірки проєктами C# коду
SettingsPage	Клієнтський клас, що містить в собі логіку та користувацький інтерфейс для керування налаштуваннями користувача та системи, зокрема основного алгоритму
Tree	Клас даних, що необхідний для збереження структури абстрактного синтаксичного дерева
TreeNode	Клас даних, що необхідний для збереження частини структури абстрактного синтаксичного дерева та містить в собі дані про вузол, тобто атомарну частину синтаксису

Продовження таблиці 4.1

Клас	**Опис**
NotificationDto	Клас даних, що містить налаштування для відправлення нотифікацій користувачу системи
Analyzed<T>	Клас даних, який представляє додатковую до вхідної структури, що містить результати аналізу та простановки маркеру
AnalyzerResult	Клас даних, що містить всі дані про результат обробки алгоритмом, може в подальшому бути перетвореним в різні представлення
Marker	Клас даних, що містить інформацію про типізований маркер та про те, який в даний момент вид типу він має
ClassModel	Клас даних, що необхідний для збереження структури семантичної моделі класу
MethodModel	Клас даних, що необхідний для збереження структури семантичної моделі дерева
Project	Клас даних, що необхідний для збереження семантичної, файлової та синтаксичної структруи оброблюваного проєкту

4.3.2 Опис методів

Методи класів наведено в таблиці 4.2.

Таблиця 4.2 – Таблиця методів

Найменування класу	**Сигнатура методу**	**Опис**
ProjectController	Project: fileStream -> IApiResult	Завантаження проекту для аналізу
ProjectController	ViewAll: void -> IApiResult	Перегляд всіх завантажених проектів в системі
ProjectController	Pause: string -> IApiResult	Пауза процесу алгоритму по етапам
SettingsController	UpdateSettings: Settings -> IApiResult	Оновленя налаштувань
SettingsController	GetSettings: void -> IApiResult	Повернення налаштувань, відповідно до налаштованого проекту користувача
SettingsController	CreateUser: UserDto -> IApiResult	Створення користувача в системі
SettingsController	Reset: void -> IApiResult	Відновлення даних налащтувань
NotificationController	AddNotification: NotificationDto -> IApiResult	Додати нотифікацію про результат
NotificationController	RemoveNotification: string -> IApiResult	Видалати нотифікацію з API користувача для видалення

Продовження таблиці 4.2

Найменування класу	**Сигнатура методу**	**Опис**
NotificationController	ListNotificationKinds: void -> IApiResult	Отримати список доступних різновидів нотифікацій
ProjectRepository	StoreProject: Stream -> StoreOptions	Зберегти проект в базі даних
ProjectRepository	RemoveProject: string -> void	Видалити проект з бази даних
ProjectRepository	AddTypings: ProjectTree -> void	Оновити структуру проекту в базі даних шляхом додавання даних про типи
SettingsRepository	AddSettings: Settings -> void	Додати нові налаштування в базу даних
SettingsRepository	RemoveSettings: string -> void	Видалити налаштування для користувача з бази даних застосунку
SettingsRepository	UpdateSettings: Settings -> void	Оновити налаштування в базі даних
AbstractSyntaxTreeParser	Parse: void -> AstModel	Отримати AST стуктуру для коду
AbstractSyntaxTreeParser	ParseClass: void -> ClassModel	Розібрати структуру класу
AbstractSyntaxTreeParser	ParseBody: void -> BodyStructure	Вивести тіло функції в структуру дерева

Продовження таблиці 4.2

Найменування класу	**Сигнатура методу**	**Опис**
AbstractSyntaxTreeParser	ParseExpression: void -> ExpressionModel	Отримати структуру виразу з методу, що аналізується
AbstractSyntaxTreeParser	ParseStatement: void-> StatementModel	Парсинг інструкції в вихідному коді методу, що належить класу, що аналізується
AbstractSyntaxTreeParser	FetchMethdoReferences: string -> Reference[]	Отримати з семантичної моделі всі прям та поліморфні посилання на вказаний метод
AlgorithmWalker	WalkClass: ClassModel -> Analyzed<ClassModel>	Робить прохід по моделі класу
AlgorithmWalker	WalkMethod: MethodModel -> Analyzed<MethodModel>	Виконує прохід по моделі методу
AlgorithmWalker	AddTypeLabel<T>: T -> Analyzed<T>	Виконує роботу алгоритму по простановці типізованої мітки в одиниці коду
AlgorithmWalker	MergeTypeLabels<TLeft, TRight, TResult>: (TLeft, TRight) -> TResult	Виконує об’єднання типізованої мітки згідно проходу алгоритму по різним гілкам виконання коду програмного забезпеченя

Продовження таблиці 4.2

Найменування класу	Сигнатура методу	Опис
AlgorithmWalker	BuildTypizedMarker: (TypeKind, Instruction) -> Marker	Виконує додавання маркеру до об’єкту інструкції в методі, що аналізується
AlgorithmWalker	FindPolymorphicAlternatives: void -> TreeNode[]	Пошук всіх поліморфних методів для аналізованого класу
AlgorithmWalker	FindStartingMethod: void -> TreeNode[]	Пошук всіх місць коду, що викликаються ззовні а також методу Main
AlgorithmWalker	MergePolymorphicTypes: Marker[] -> Marker	Об’єднує можливі значення результату поліморфного методу та бере найслабіший фільтр для типу
AlgorithmWalker	FindIdentificators: void -> TreeNode[]	Пошук всіх ідентифікаторів в поточному методі
AlgorithmWalker	SaveMethodScheme: void -> void	Зберігає тимчасову схему даних до бази даних
ProjectService	StartAlgo: Tree -> Tree	Виконує старт алгоритму статичного аналізу
ProjectService	PauseAlgo: void -> void	Ставить на паузу код

Продовження таблиці 4.2

Найменування класу	**Сигнатура методу**	**Опис**
ProjectService	StopAlgo: void -> void	Виконує повну зупинку виконання, зберігає в стані класу результати роботи
ProjectService	DemonstrateResult: void -> AnalyzeResult	Виконує повернення збережених в стані класу результатів роботи алгортму
NotificationService	PushUserNotification: NotificationDto -> void	Проводить надсилання данних про аналі по вказаним параметрам нотифікацій
ProjectLoader	LoadZipFile: Stream -> Project	Проводить завантаження з zip файлу в модель, з якої можна отримати файли
ProjectLoader	SelectFile: string - > string	Виконує отримання контенту файлу з архіву
ProjectLoader	FindProjectReferences: void -> Reference[]	Отримати залежності проекту на інші проекти в Microsoft Solution File
ProjectLoader	PackArchive: Project -> Stream	Виконує зворотню запаковку проекту з результатами аналізу систем

Продовження таблиці 4.2

Найменування класу	**Сигнатура методу**	**Опис**
Utils	RemoveSpaces: string -> string	Видалити всі зайві пробіли зі заданої строки
Utils	MakeJson: object -> string	Створює строку в форматі JSON який повторює типізовна струткту ру вхідного об’єкта
Utils	FindNode: (Tree, string) -> Node	Пошут елемента в дереві абстрактної синтаксичної структури коду
Utils	AttachType: (Node, Type) -> Node	Додає типізовану мітку в структуру даних абстрактного синтаксичного дерева коду
Utils	FindSubNodesByName: (Node, string) -> Node[]	Виконує пошук по дереві в глубину для знаходження елемента з заданим іменем
LoginPage	Login: (string, string) -> Promise	Виконує вхід в систему
ProjectPage	LoadProject: void -> Promise	Виконує завантаження проекту до сторінки

Продовження таблиці 4.2

Найменування класу	**Сигнатура методу**	**Опис**
ProjectPage	ViewUpdatedInformation: void -> void	Робить установку для перегляду оновленої інформації проекту згідно обробленої частини
ProjectPage	Refresh: void -> void	Виконується отримення оновлених даних про аналізу коду с серверної частини
SettingsPage	SetupSetting: Setting -> void	Виконується оновлення вибраного налаштування
SettingsPage	SendSettingsToServer: void -> Promise	Виконує надсилання останніх оновлених налштувань користувача до серверної частини сервісу
SettingsPage	ResetDefaults: void -> Promise	Виконує установку стандартних налаштувань для застосунку та надсилає дані до сервісу

4.3.3 Діаграма компонентів

Поділимо отримане програмне забезпечення на набір компонентів-збірок на рисунку 4.5.

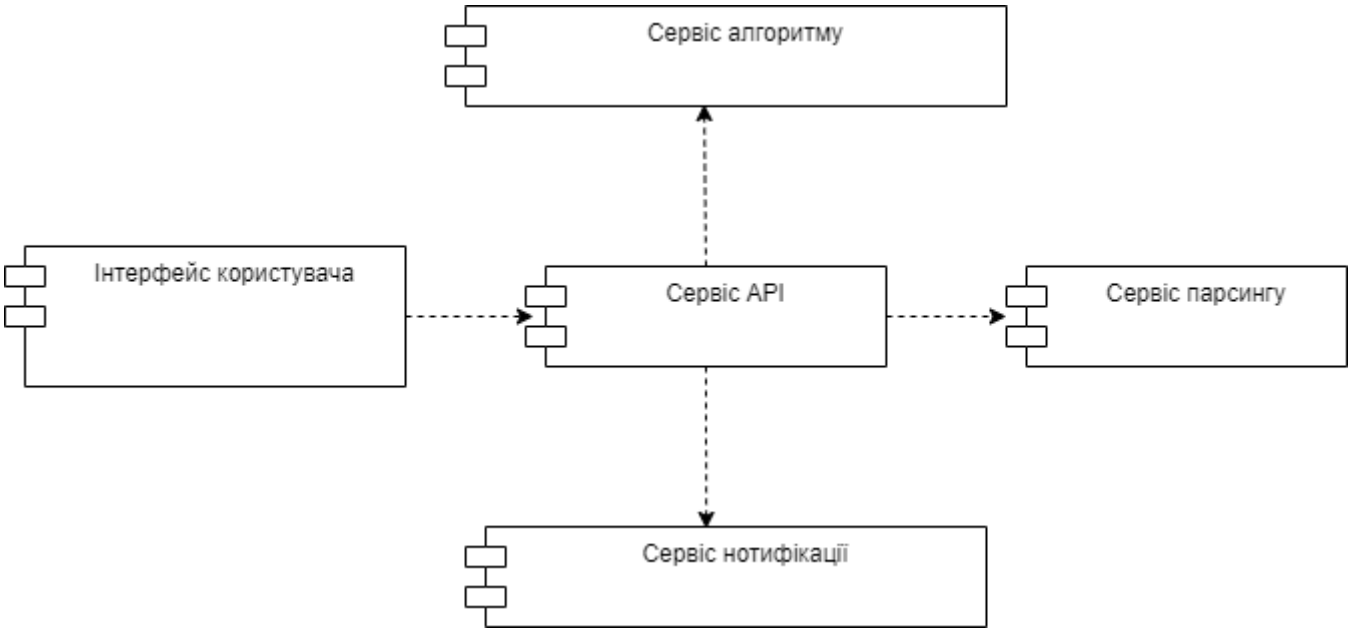


Рисунок 4.5 – Діаграма компонентів

Програмне забезпечення поділено на декілька частин в залежності від використовуваної технології та бібліотек, що наявні, та представляють функціональне вирішення окремої задачі. Компонентна архітектура програмного забезпечення надає можливості повторноговикористання коду, масштабування, та забезпечує економію часу при подальшій роботі інженерів програмного забезпечення.

Висновки до розділу

В даному розділі було розроблено архітектуру проєкту, обрано компоненту модель, яка включає можливість поділу на незалежні бібліотеки, досліджено та обгрунтовано технології, що будуть використовуватись в розробці алгоритму, що забезпечує вирішення задач.

Також було описано набір бібліотек, що використовувася, підкреслено їх сильні та слабкі сторони, виведено ті риси, які придатні для розробки системи.

5 МОДИФІКОВАНИЙ АЛГОРИТМ СТАТИЧНОГО АНАЛІЗУ

Даний розділ містить аналіз відомих алгоритмів та синтез нового алгоритму, що має наукову новизну. Також буде проведено порівняльний аналіз з іншими алгоритмами. В якості опорного методу для статичного аналізу був вибраний алгоритм data-flow. Цей метод має ряд недоліків, зокрема він не може працювати з інкапсуляцією та поліморфним кодом, погано виводить типи для змінних після умовних конструкцій. Тому було вирішено використати інші методи, зокрема і ті, що вирішують дещо інші задачі, але містять ряд підходів, що зможуть вирішити недоліки опорного алгоритму. Зокрема було вирішено взяти структуру даних, що містить стан виконання для кожної інструкції в виїхідному коді, з алгоритму структурного аналізу в мові Typescript, а також механізм простановки прапорців з алгоритму taint-аналізу.

Використані елементи дають змогу аналізувати код, що містить інкапсульовані дані, додаючи інформацію про стан об’єкту в структуру даних інструкції, а також обходити поліморфізм за допомогою підходу об’єднання типів, що було взято з Typescript. Також було запозичено лексеми any, unknown, never, які є узагальненими та базовими в системі типів.

5.1 Опис розроблюваного алгоритму

Отже, суть модифікованого алгоритму в вводу додаткових типізованих міток до кожної змінної в кожній інструкції, змінна може бути як примітивом, так і об’єктом з інкапсуляцією, в такому випадку для узагальнення будь-яка змінна вважається як об’єкт зі станом, при чому примітивні типи містимуть лише одне поле – значення цієї змінної. Крім того, тип може мати 4 різновиди:

- конкретне значення (const);
- будь-яке значення в межах типу (any);
- набір можливих значень (union);
- умовний (condition).

Такий вибір поділу на типи дає перевагу над іншими методами, оскільки дає змогу включати умовні оператори, поєднувати декілька гілок коду, за допомогою union, визначати тип, як той, що може містить будь-які значення, що дає змогу моделювати дані, що прийшли ззвоні, цей елемент використовує можливості taint-аналізу. Також доступний константний тип, що дозволяє моделювати ситуації, коли інструкцій в коді завжди буде виконуватись при певному значенні змінної, що надає можливість виявляти гілки коду, які можуть бути не виконаними взагалі. Також такий тип здатен сигналізувати про помилкові ситуації при доступі до індексаторів.

Відобразимо модифікований алгоритм схематично, вважаємо, що початково побудований граф викликів:

а) отримати всі методи, на які немає посилань та Main;

б) виставити вхідні параметри як typeName:any;

в) покрокових прохід по функції;

1) якщо створення константи – установити тип const;
2) якщо умовна перевірка – уточнення стану згідно умови (використавши попередньо обчисленні типізовані мітки);
   – для кожної інструкції перевірки – застосувати умовний обмежувач на значення condition;
3) якщо неполіморфна функція перетворення – аналіз функції (перейти до пункту 3);
4) якщо поліморфна фунцкія перетворення;
   – знайти всі можливі поліморфні імплементації;
   – для кожної імплементації – перехід до пункту 3;
   – поєднання типізованих умов в найменш слабіше;

г) вихід.

Алгоритм представлений для ітерації поверх одного методу, прото сама структура викликів в програмі представляє граф, причому циклічний, також кожна із умов включає в себе складну перевірку конкретних випадків умов, типів та констант. Все це і складає його основну складність.

Слід зазначити, що алгоритм в себе включає механізм обходу поліморфізму за допомогою 2 стратегій:

– конкретний код, що будет запускатись відомий статично;

– код, що буде запускатись, залежить від вхідних даних, які не керовані алгоритмом.

В першому випадку буде виводитись конкретний тип, що повертає відомий статично код, що буде запускатись, водночас другий буде проходити через механізм об'єднання типізовних значень статичних методів.

5.2 Розроблена структура даних для оперування

Наведемо елементи розробленої структури даних AST, що забезпечує реалізацію тих функцій, які існують в опорному алгоритму, та тих, які додані з додаткових алгоритмів, в таблиці 5.1.

Таблиця 5.1 – Структура AST

Назва елементу	**Структура елементу**

project	{
	“documents”: <document>[]
	}

document	{
	“namespace”: string,
	“items”:
	<namespaceMember.class |
	namespaceMember.interface>[]
	}

Продовження таблиці 5.1

Назва елементу	**Структура елементу**

namespaceMember: class	{
	“kind”: “class”,
	“name”: string,
	“modifiers”: string[],
	“base”: string[],
	“body”: <classMember>[],
	“astStructureMode”: string[]
	}

method	{
	“kind”: “method”,
	“name”: string,
	“returns”: <expression>,
	“parameters”: <parameter>[],
	“body”: <expression>
	}

parameter	{
	“type”:<typeSyntax>,
	“name”: string
	}

statement	{
	“kind”: “statement.X “,
	“type”: <typeSyntax>,
	“name”: string,
	“expression”:<expression>
	}

Продовження таблиці 5.1

Назва елементу	**Структура елементу**

expression	{
	“kind”: “expression.X”,
	“left”: <expression>,
	“right”: <expression>,
	“operator”: string
	}

property	{
	“kind”: “classMember.property”,
	“type”: <typeSyntax>,
	“name”: string,
	“modificators”: string[],
	“accessors”: string[],
	“body”:<expression> | <statement>[]
	}

field	{
	“kind”: “classMember.field”,
	“type”: <typeSyntax>,
	“variables”: <variableDeclaration>,
	“modificators”: string[]
	}

event	{
	“kind”: “classMember.event”,
	}

Продовження таблиці 5.1

Назва елементу	Структура елементу

typeSyntax	{
	“kind”: “type.X”,
	“name”:string
	}

variableDeclaration	{
	“kind”: “variable “,
	“name”:string,
	“type”: <typeSyntax>
	}

5.3 Порівняльний аналіз

Щодо переваг саме цього модифікованого алгоритму можна віднести:

– перевірка об’єктів зі станом(зміна фактичного типу для кожної інструкції);

– аналіз поліморфних викликів (перебір можливих значень серед існучих імплеметацій);

– можливий звичайний вивід можливих типів для аналізу розробником.

Слід зазначити, що алгоритм містить досить універсальну структуру даних, тобто його результати можна використовувати згодом для аналізу SQL-ін’єкій, що робить taint-аналіз (перевіривши в місці передачі назовні any<string>, наприклад), а також для аналізу коду, який ніколи не виконається, чи умови, що ніколи не виконається.

Модифікований алгоритм дозволяє знаходити більшу кількість помилкових ситуацій, завдяки покращеній та узагальненій структурі даних. Конкретний приклад виводу представлений на рисунку 5.1.

```
    Dictionary<string, string> GetFromDb() {
        var result = Load(); // {[Key extends string]: [Value extends string]}

        result["Source"] = "ProjectName"; // {[Key extends string]: [Value extends string], "Source": "ProjectName"}

        return result;
    }

    string GetServiceOutput()
    {
        var dbResult = GetFromDb(); // {[Key extends string]: [Value extends string], "Source": "ProjectName"}

        string content = $"Service name: {dbResult["Source"]}"
            + $"Description: {dbResult["Description"]}"
            + $"Row data: {string.Join(", ", dbResult.Select(x => x.Key + ":" + x.Value))}"; // помилка Description не є гарантованим

        return content;
    }
}
```

Рисунок 5.1 – Результат аналізу за допомогою модифікованого методу

В даному випадку звичайний data-flow аналіз не зможе визначити помилкову ситуацію, наведену на рисунку, проте модифікований алгоритм завдяки можливості аналізувати інкапсульовані об’єкти здатен зафіксувати, що змінна dbResult на містить ключа Description, та повідомить про помилкову ситуацію.

5.4 Обчислювальна складність алгоритму

Часові затрати складаються з таких елементів:

– парсинг вхідного коду;

– робота алгоритму;

– представлення результату.

Перший та третій пункти є такими, що час їх виконання майже не впливає на виконання всього циклу програми, тому не будемо виконувати обчислення їх часу запуску. Детально зупинимось на другому пункті. Тобто заміри будемо проводити на роботі лише алгоритму.

Кожна програма може давати різне значення в результаті і це залежить від таких умов:

– розмір проєкту;

– цикломатична складність;

– наявність поліморфних викликів.

Поділимо по першому показнику програми умовно на 3 типи:

– великі, містять більше 1000 строк;

– середні, містять100 – 1000 строк;

– малі, містять до 100 строк.

З огляду на те, що програмне забезпечення призначене для аналізу лабораторних робіт, умови застосування, а саме навчання та розробка маємо необхідність у вивчені всіх видів проєктів, проте зупинимось лише на малому та середньому, оскільки великі проекти можуть мати великі погрішності, бо в цьому випадку час роботи алгоритму залежить більше від саме того, як побудовний застосунок за дизайном.

Вважатимемо, що робота алгоритму маштабується з лінійною залежністю від саме розміру проєкту (ускладнення складають циклічні залежності та поліморфізм). Поділимо експеримент на 5 тестів, в кожному з яких буде по 100 запусків та обчислене середнє значення. На рисунку 5.2 зображено час запуску алгоритму для середнього за розміром проєкту при 100 спробах.

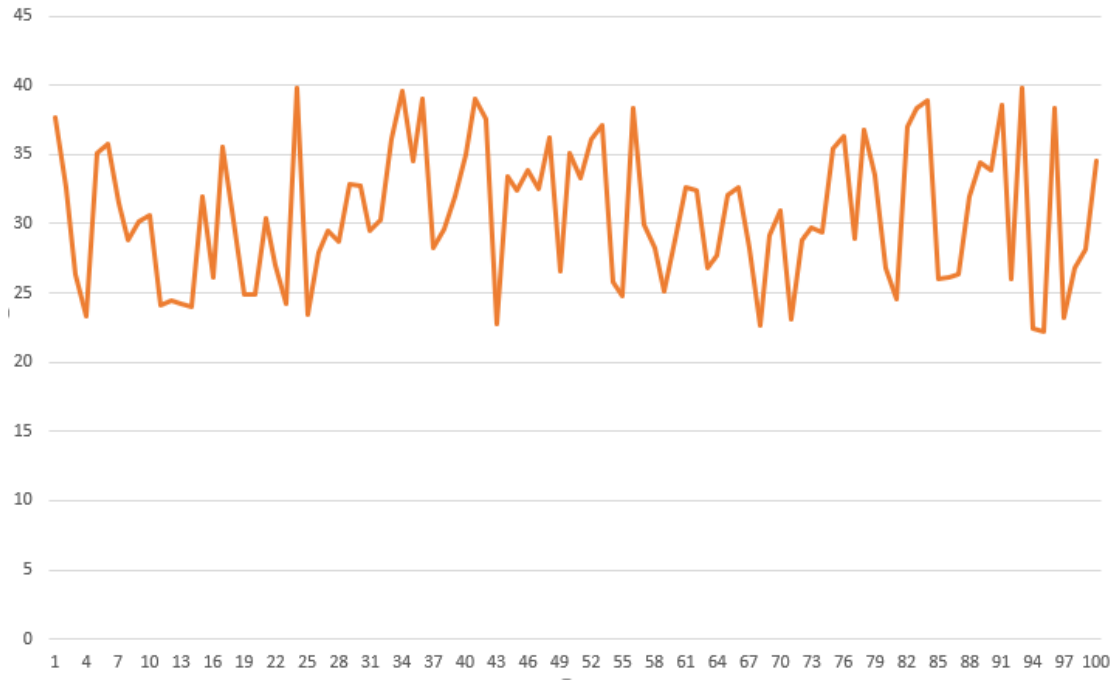


Рисунок 5.2 – Час виконання для 100 запусків

В таблиці 5.2 представлені результати тестових запусків. Час відображений в секундах.

Таблиця 5.2 – Тестові запуски алгоритму*

Тип	**Тест 1**	**Тест 2**	**Тест 3**	**Тест 4**	**Тест 5**	**Середній час по тестах**
Малий	7.52 с	6.22 с	6.13 с	6.15 с	6.12 с	6.43 с
Середній	29.01 с	31.17 с	35.98 с	28.04 с	28.56 с	30.55 с

* - наведено середній час запуску кожного тесту зі 100 спроб.

Як видно з даних таблиці розкид часу складає близько 20%, що пояснюється відхиленнями в навантаженні CPU та пріоритетах процесів.

Висновок до розділу

Таким чином був синтезований алгоритм, основною новизною якого є введення типізованих міток для верифікації короткності вихідного коду. Були виділені характерні риси алгориму, представлена його логічна структура та модель, якою він оперує. Також був заміряний час його виконання на типових прикладах. Проте часова складність не є основним місцем покращення алгоритму, лише якість виведених даних, натомість час необхідний для розуміння порядку затраченого часу для роботи над різними прикладами вхідних даних.

Новостворений алгоритм має переваги над існуючими та може бути застосований як окремий компонент будь-якого програмного забезпечення, зокрема системи, опис якої буде представлений в наступному розділі.

Слід зазначити, що новостворений алгоритм є універсальним та може бути використаним для отримання звітів по іншим типам статичної обробки тексту.

6 РОЗРОБЛЕННЯ СТАРТАП ПРОЕКТУ

6.1 Опис ідеї проекту

Отже, проведемо аналіз цього проєкту на можливість бути стартапом та конкурувати з іншими проєктаму, в тому числі і з аналогами.

Завданням проєкту є покращення процесу розробки шляхом верифікації програмного забезпеченн, а також здійснити покращення навчального середовища, шляхом створення системи, що надає відгуки щодо якості коду.

Суть полягає в створенні засобу, який дозволяє проводити верифікування вихідного коду та виводити обраховані риси щодо складності, надійності.

Цільова аудиторія проєкту – розробники та студенти, що бажають покращити якості власного ПО та отримувати зворотній відгук про результати своєї роботи в автоматичному режимі.

Вигода використання продуту полягає в покращеному процесі розробки та навчання.

Щоб мати цілісну уяву стосовно можливостей проєкта на ринку, його конкурентоспроможності відобразимо деталі ідеї в таблиці 6.1

Таблиця 6.1 – Ідеї проєкту

Зміст ідеї	**Напрямки застосування**	**Вигода користувача**
Система статичного аналізу вихідного коду на основі системи виводу типів Хіндлі-Мілнера для об’єктів зі станом	Розробка програмного забезпечення, навчання в сфері інформаційних технологій	Автоматична перевірка результатів кодування та відображення вразливих місць програми.

Таким чином ідея статичного аналізу коду є актуальною, оскільки ринок розробки росте все більше та більше,а також збільшується необхідність подібних інструментах в системах з критичними показниками виконання.

Також в систему була додана можливість нотифікацій та прості CI інтеграції, що необхідно для зручності користувача.

Далі слід проаналізувати конкуруючі рішення на ринку, для того, щоб визначити, чи має цей стартап можливість функціонально вирізнятись. Введемо позначення відносних характеристик:

- більш слабкі значення (W);
- аналогічні значення (N);
- сильні значення (S) .

Після пошуку анологічних інстурментів було знайдено SonarQube.

У таблиці 6.2 наведений аналіз та порівняння рішень.

Таблиця 6.2 – Порівняння за аналогами

Фактична ідея	**Проєкти**		**W, слабке значення**	**N, нейтральне**	**S, сильне значення**
	Поточний	**SonarQube**			
Поверхневий аналіз				+	
Інтеграція з різними системами		+	+		
Глибокий аналіз потоку даних	+				+
Пошук помилкових ситуацій	+				+

Проаналізувавши порівняльну таблицю можна зробити деякі висновки, щодо спроможності проєкту до конкуренції. Зокрема сильними сторонами можна вважати

якість аналізу, а саме його глибина та можливість виводити помилкові ситуації. На відміну від SonarQube система може видавати не тількі поверхневі показники, що лише частково впливають на якість коду, а саме його процес виконання.

6.2 Технічний аудит проєкту

Цей розділ посвятимо пошуку методів, що сприяють запуску стартапу. Розглянемо технології проєкту та відобразимо в таблиці 6.3.

Таблиця 6.3 – Технології з проєкту

Ідея	**Технології**	**Наявність технологій в середовищі**	**Доступність**
Фронтенд частина додатку	React, Typescript	Є	Доступна та безкоштовна
Сервіс управління верифікаціями	Typescript, Node.JS	Є	Доступна та безкоштовна
Алгоритм отримання дерева	C#	Є	Доступний та безкоштовний
Алгоритм верифікації	Typescript, Node.JS	Є	Доступний та безкоштовний

Таким чином, використовувані технології базуються на відкритих та безкоштовних джерелах.

6.3 Аналіз ринку для запуску проєкту

Визначимо ринкові можливості, та проблеми, що можуть спіткати проєкт при виході на ринок. Слід зауважити, що цей ринок є багатогранним, а отже і може включати певну множину непов'язаних індикаторів. Відобразимо це в таблиці 6.4

Таблиця 6.4 – Аналіз ринку

Номер	Індикатор	Значення
1	Кількість основних гравців	2
2	Загальна сума продажів, грн/ум.од	Невідома, поширено в 30% ринку
3	Динаміка оновлення показників ринку	Зростає
4	Вхідні обмеження	Немає
5	Стандартизація та сертифікація	Можливі для деяких видів безпекових верифікаторів
6	Середня норма рентабельності	Невідомо

Як висновок, проєкт має значні можливості для виходу на ринок, оскільки не покрита ще значна частина ринку, скоріше за все необхідно акцентувати увагу на якісних рисах проєкту.

6.4 Маркетингова програма стартап-проєкту

Проведемо формування маркетингової програми для ринку. Застосунок буде поширюватись в 2 видах: Community Edition та Enterprise Edition, що є традійним підходом до поділення на види за оплатністю. Такий підхід дозволяє отримувати більше користувачів базовою версієї, та готувати їх да платної версії, показавши її переваги. Основним обмеженням буде виступати – можливість деталей налаштувань, кількість аналізів в день.

Зазначимо основні плюси використання продукту, а також вигоду, яку отримує кінцевий користувач додатку разом з перевагами використання сервісу створеного в рамках проєкту саме цього стартапу в таблиці 6.5, а також проаналізуємо результати

виходячи з переваг та вигідності використання для користувача, а також його потреб в системі статичного аналізу коду.

Таблиця 6.5 – Принципи використання продукту

Потреба	**Вигідність при використанні**	**Переваги**
Глибокий аналіз на можливі помилкові ситуації	Запобігання великої кількості помилок, що йдуть до продуктового середовища	Глибокий аналіз помилок

Тобто продукт має специфічну перевагу над конкурентами, що значно посилює його перевагу на ринку, також слід проаналізувати продажі. Для цього зобразимо систему продаж в таблиці 6.6 та наведемо обгрунтування вибору того чи іншого методу продажів.

Таблиця 6.6 – Збут

Специфіка закупівель	**Функції збуту**	**Заглибленість каналу продаж**	**Найкраща система продаж**
Замовлення	- Розробка - Тестування - Додавання функціоналу - Продажі та підтримка	Перший рівень	Пряма взаємодія з клієнтами

Необхідно мати стратегію комунікацій, для цього слід виділити декілька показників, які впливають на можливість здійснення комунікації, а також поведінкову модель клієнтів, також виведемо рекламне звернення, що має бути

застосовано, результати цього аналізу зобразимо в таблиці 6.7, а також визначимо ключові чинники, но основі яких має будуватись стратегія комунікацій проєкту відповідно до аудиторії.

Таблиця 6.7 – Поведінка клієнтів

Особливості поведінки клієнтів	**Канали комунікації**	**Основні властивості для позиціювання**	**Рекламне звернення**
Збільшення надійності створюваного ПО	Інтернет, реклама, статті	Канал першого рівня	Збільшення глибини та якості аналізу логіки коду

Як бачимо, поведінка клієнтів більше базується на якостях даного проєкту.

6.5 Ринкова стратегія проєкту

Зобразимо потенційних користувачів в таблиці 6.8

Таблиця 6.8 – Аналіз потенційних користувачів

Група	**Готовність до користування**	**Попит**	**Конкуренція**	**Складність входу**
Розробники програмного забезпечення	Готові	Високий	Висока	Середня
Учасники навчального процесу в сфері розробки	Готові	Середній	Середня	Висока

Відповідно до таблиці, ринкова стратегія продукту має орієнтуватись на 2 типи користувачів, при чому вони мають різні підходи до впровадження через різні можливості конкуренції та попит. Як висновок, слід орієнтуватись на гарну рекламу серед другої групи та акцент на кращих властивостей продукту для першої групи.

Також наведемо стратегію розвитку в цільових напрямках в таблиці 6.9 та зробимо висновки відносно перспектив розвитку.

Таблиця 6.9 – Стратегії виходу на ринок

Альтернатива розвитку	**Стратегія охоплення**	**Конкурентноспроможні позиції альтернативи**	**Основний спосіб розвитку**
Спеціалізація	Реклама, удосконалення ПО	Якісний продукт	Диференціація

Отже, за основу можна взяти стратегію диференціації — тобто орієнтацію на користувача. Якщо цей підхід не спрацює буде застосована спеціалізація для конкретної групи людей.

З'ясуємо стратегію конкурентної поведінки на ринку, та відобразимо у таблиці 6.10.

Таблиця 6.10 – Конкурентна поведінка

Чи є проект першопрохідцем?	**Шукати нових користувачів чи забирати існуючих?**	**Чи буде копіювати існуючі характеристики товару конкурента?**	**Стратегія конкурентної поведінки**
Ні	Завойовувати існуючих	Ні	Зайняття конкурентної ніші

Отже, обраною стратегією є зайняття конкурентної ніші з охопленням декількох сегментів споживачів.Продукт не є першопрохідцем на ринку, тобто потрібно буде забирати клієнтів у існуючих конкурентів.

Зобразимо стратегію позиціюваня в таблиці 6.11 та проведемо аналіз моделі позиціювання на основі результатів агрегування даних в таблиці.

Таблиця 6.11 - Позиціювання

Вимоги до товару цільової аудиторії	**Базова стратегія розвитку**	**Ключові конкурентноспроможні показники**	**Вибір асоціацій, які мають сформувати комплексну позицію власного проекту**
Удосконалення механізмів аналізу вихідного коду. Якісний зворотній відклик сервісу стосовно рис коду.	Диференціація	Глибина аналізу, більше охоплення коду	Якість Надійність Точність

Таком чином, базовою стратегію розвитку є диференціація.

Висновки до розділу

Головним завдання розділу є створення комплексного підходу до позиціювання та росповсюдження проєкту з метою отримати якнайширшу аудиторію з доброзичливими відгуками про продукт. Аналоги продукту не виконують те, що

виконує цей алгоритм в повній мірі, тому цей продукт може мати успіх в разі розуміння клієнтів, що його використання призводить до більшого доходу, в результаті уникання ситуацій з неякісним кодом.

Було проводено аналіз можливих ризиків та сформульований альтернативний підхід, що дозволить продукту мати більш стабільну клієнстьку базу.

До того ж побудовано маркетинговий план та визначено стратегії розвитку продукту на ринку.

ВИСНОВКИ

При виконанні магістерської дисертації вирішувалась задача удосконаленння статичного аналізу тексту, а саме аналізу потоку даних. Проблематика полягає в тому, що існуючі методи не можуть давати достовірні результати при інкапсульованому підході до побудови об'єктів, а також при використанні поліморфізму. Було розглянуто декілька вже відомих алгоритмів статичного аналізу, кожен з яких має як переваги так і недоліки, які були розглянуті та описані. Ні один з існуючих алгоритмів не дає рішення задачі статичного аналізу у тому обсязі, у якому його вирішує синтезований алгоритм.

Розроблений алгоритм є модифікацією алгоритму data-flow аналізу, яка полягає в зміні структури даних для кожної інструкції, що дозволяє обробляти код з інкапсульованими та поліморфними об’єктами. Було застосовано як існуючі підходи, що вирішують задачу аналізу коду, так і рішення які пов’язані з цією проблемою лише частково, зокрема система виводу типів та структурний аналіз. Було зазначено, чим саме дані методи допомагають у вирішенні проблеми. Розроблений алгоритм є синтезом із вже існуючих та використовує додаткові підходи у вирішенні поставленої задачі аналізу коду.

Запронований метод складається з елементів аналізу потоку даних та алгоритму виводу типів Хіндлі-Мілнера. В якості вхідних параметрів він приймає програмний код на мові C#, проте, за своєю структурою, є універсальним і придатний до застосування для інших мов із C-подібний синтаксисом та з об'єктно-орієнтованими можливостями.

Огляд існуючих підходів до статичного аналізу та пошуку характерних рис програмного коду були описані в розділі «Аналіз методів статичного аналізу коду».

Модифікований алгоритм, а також модель даних були описані в розділі «Модифікований алгоритм статичного аналізу».

На основі запропонованого методу створене програмне забезпечення для статичного аналізу коду. Проведено якісний аналіз коду та обчислювальної складності алгоритму для коду різних розмірів. Розкид часу для проектів різних

розмірів не перевищє 20%. Загальний час виконання для середніх та малих проектів лежить в межах 30 секунд.

За результатами дисертації було опубліковано тези до конференції.

ПЕРЕЛІК ПОСИЛАНЬ

1) Wichmann, B. A.; Canning, A. A.; Clutterbuck, D. L.; Winsbarrow, L. A.; Ward, N. J.; Marsh, D. W. R. (Mar 2018). "Industrial Perspective on Static Analysis"

2) Egele, Manuel; Scholte, Theodoor; Kirda, Engin; Kruegel, Christopher (2019-03-05). "A survey on automated dynamic malware-analysis techniques and tools"

3) Achim D. Brucker and Uwe Sodan. (2016) Deploying Static Application Security Testing on a Large Scale.

4) Oh, Hakjoo; Yang, Hongseok; Yi, Kwangkeun (2016). "Learning a strategy for adapting a program analysis via bayesian optimisation".

5) Monperrus, Martin; Mezini, Mira (2017). "Detecting missing method calls as violations of the majority rule"

6) Грэди Буч. Объектно-ориентированный анализ и проектирование с примерами приложений на C++ = Object-Oriented Analysis and Design with Applications / Пер. И.Романовский, Ф.Андреев. — 2-е изд. — М., СПб.: «Бином», «Невский диалект», 1998. — 560 с.

7) Вивід типів [Електрониий ресурс] – 2020. – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/%D0%92%D0%B8%D0%B2%D1%96%D0%B4_%D1%82%D0%B8%D0%BF%D1%96%D0%B2

8) Matt Weisfeld. The Object-Oriented Thought Process. — Fourth Edition. — Addison-Wesley Professional, 2013. — 336 с.

9) Bartel, Alexandre; Klein, Jacques; Monperrus, Martin; Le Traon, Yves (1 June 2014). "Static Analysis for Extracting Permission Checks of a Large Scale Framework: The Challenges and Solutions for Analyzing Android".

10) Kildall, Gary Arlen (May 1972). Global expression optimization during compilation.

11) Bodden, Eric (2012). "Inter-procedural data-flow analysis with IFDS/IDE and Soot". Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis.

12) Rüthing, Oliver; Knoop, Jens; Steffen, Bernhard (2003-07-31). "Optimization: Detecting Equalities of Variables, Combining Efficiency with Precision".

13) Mohnen, Markus (2002). A Graph-Free Approach to Data-Flow Analysis. Lecture Notes in Computer Science.

14) Cousot, Patrick; Cousot, Radhia (1977). "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints".

15) Faure, Christèle. "PolySpace Technologies History".

16) Philippe Granger (1991). "Static Analysis of Linear Congruence Equalities Among Variables of a Program".

17) Michael Karr (1976). "Affine Relationships Among Variables of a Program".

18) Bennett, J. M.; Prinz, D. G.; Woods, M. L. (1952). "Interpretative sub-routines".

19) MongoDB [Електронний ресурс] – 2020. – Режим доступу до ресурсу: https://www.mongodb.com

20) Fastify [Електронний ресурс] – 2020. – Режим доступу до ресурсу: https://www.fastify.io/

ДОДАТОК А
ЛІСТИНГ ПРОГРАМИ

DataVerification.cs

```
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;
using Microsoft.CodeAnalysis.Diagnostics;
using System.Collections.Generic;
using System.Collections.Immutable;
using System.Linq;
using CodeCracker.Properties;

namespace CodeCracker.CSharp.Maintainability
{
    [DiagnosticAnalyzer(LanguageNames.CSharp)]
    public sealed class XmlDocumentationAnalyzer : DiagnosticAnalyzer
    {
        internal static readonly LocalizableString Title = new
LocalizableResourceString(nameof(Resources.XmlDocumentationAnalyzer_Title),
Resources.ResourceManager, typeof(Resources));

        internal static readonly DiagnosticDescriptor RuleMissingInCSharp = new
DiagnosticDescriptor(
            DiagnosticId.XmlDocumentation_MissingInCSharp.ToDiagnosticId(),
            Title,
            Title,
            SupportedCategories.Maintainability,
            DiagnosticSeverity.Info,
            isEnabledByDefault: true,
            helpLinkUri:
HelpLink.ForDiagnostic(DiagnosticId.XmlDocumentation_MissingInCSharp));

        internal static readonly DiagnosticDescriptor RuleMissingInXml = new
DiagnosticDescriptor(
            DiagnosticId.XmlDocumentation_MissingInXml.ToDiagnosticId(),
            Title,
            Title,
            SupportedCategories.Maintainability,
            DiagnosticSeverity.Warning,
            isEnabledByDefault: true,
            helpLinkUri:
HelpLink.ForDiagnostic(DiagnosticId.XmlDocumentation_MissingInXml));

        public override ImmutableArray<DiagnosticDescriptor> SupportedDiagnostics =>
ImmutableArray.Create(RuleMissingInCSharp, RuleMissingInXml);

        public override void Initialize(AnalysisContext context) =>
context.RegisterSyntaxNodeAction(Analyzer,
SyntaxKind.SingleLineDocumentationCommentTrivia);

        private static void Analyzer(SyntaxNodeAnalysisContext context)
        {
            if (context.IsGenerated()) return;
            var documentationNode = (DocumentationCommentTriviaSyntax)context.Node;
            var method = GetMethodFromXmlDocumentation(documentationNode);
            if (method == null) return;
            var elementNames = documentationNode.Content
                                    .OfType<XmlEmptyElementSyntax>()
                                    .Select(element =>
element.Name?.LocalName.ValueText);
            if (elementNames.Contains("inheritdoc")) return;
            var methodParameters = method.ParameterList.Parameters;
```

```
            var xElementsWitAttrs =
documentationNode.Content.OfType<XmlElementSyntax>()
                                    .Where(xEle =>
xEle.StartTag?.Name?.LocalName.ValueText == "param")
                                    .SelectMany(xEle => xEle.StartTag.Attributes,
(xEle, attr) => attr as XmlNameAttributeSyntax)
                                    .Where(attr => attr != null);

            var keys = methodParameters.Select(parameter =>
parameter.Identifier.ValueText)
                .Union(xElementsWitAttrs.Select(x =>
x.Identifier?.Identifier.ValueText))
                .ToImmutableHashSet();

            var parameterWithDocParameter = (from key in keys
                                             where key != null
                                             let Parameter =
methodParameters.FirstOrDefault(p => p.Identifier.ValueText == key)
                                             let DocParameter =
xElementsWitAttrs.FirstOrDefault(p => p.Identifier?.Identifier.ValueText == key)
                                             select new { Parameter, DocParameter
});

            if (parameterWithDocParameter.Any(p => p.Parameter == null))
            {
                var diagnostic = Diagnostic.Create(RuleMissingInCSharp,
documentationNode.GetLocation());
                context.ReportDiagnostic(diagnostic);
            }

            if (parameterWithDocParameter.Any(p => p.DocParameter == null))
            {
                var diagnostic = Diagnostic.Create(RuleMissingInXml,
documentationNode.GetLocation());
                context.ReportDiagnostic(diagnostic);
            }
        }

        private static MethodDeclarationSyntax
GetMethodFromXmlDocumentation(DocumentationCommentTriviaSyntax doc)
        {
            var tokenParent = doc.ParentTrivia.Token.Parent;
            var method = tokenParent as MethodDeclarationSyntax;
            if (method == null)
            {
                var attributeList = tokenParent as AttributeListSyntax;
                if (attributeList == null) return null;
                method = attributeList.Parent as MethodDeclarationSyntax;
            }
            return method;
        }
    }
}
```

CodeAnalysis.cs

```
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;
using Microsoft.CodeAnalysis.Diagnostics;
using Microsoft.CodeAnalysis.Formatting;
using System;
using System.Collections.Generic;
```

```
using System.Linq;

namespace CodeCracker
{
    public static class CSharpAnalyzerExtensions
    {
        public static void RegisterSyntaxNodeAction<TLanguageKindEnum>(this
AnalysisContext context, LanguageVersion greaterOrEqualThanLanguageVersion,
        Action<SyntaxNodeAnalysisContext> action, params TLanguageKindEnum[]
syntaxKinds) where TLanguageKindEnum : struct =>

context.RegisterCompilationStartAction(greaterOrEqualThanLanguageVersion,
compilationContext => compilationContext.RegisterSyntaxNodeAction(action,
syntaxKinds));

        public static void RegisterCompilationStartAction(this AnalysisContext
context, LanguageVersion greaterOrEqualThanLanguageVersion,
Action<CompilationStartAnalysisContext> registrationAction) =>
            context.RegisterCompilationStartAction(compilationContext =>
compilationContext.RunIfCSharpVersionOrGreater(greaterOrEqualThanLanguageVersion, ()
=> registrationAction?.Invoke(compilationContext)));

        public static void RegisterSymbolAction(this AnalysisContext context,
LanguageVersion greaterOrEqualThanLanguageVersion, Action<SymbolAnalysisContext>
registrationAction, params SymbolKind[] symbolKinds) =>
            context.RegisterSymbolAction(compilationContext =>
compilationContext.RunIfCSharpVersionOrGreater(greaterOrEqualThanLanguageVersion, ()
=> registrationAction?.Invoke(compilationContext)), symbolKinds);
#pragma warning disable RS1012
        private static void RunIfCSharpVersionOrGreater(this
CompilationStartAnalysisContext context, LanguageVersion
greaterOrEqualThanLanguageVersion, Action action) =>
            context.Compilation.RunIfCSharpVersionOrGreater(action,
greaterOrEqualThanLanguageVersion);
#pragma warning restore RS1012
        private static void RunIfCSharpVersionOrGreater(this Compilation
compilation, Action action, LanguageVersion greaterOrEqualThanLanguageVersion) =>
            (compilation as
CSharpCompilation)?.LanguageVersion.RunIfCSharpVersionGreater(action,
greaterOrEqualThanLanguageVersion);
        private static void RunIfCSharpVersionOrGreater(this SymbolAnalysisContext
context, LanguageVersion greaterOrEqualThanLanguageVersion, Action action) =>
            context.Compilation.RunIfCSharpVersionOrGreater(action,
greaterOrEqualThanLanguageVersion);

        private static void RunIfCSharpVersionGreater(this LanguageVersion
languageVersion, Action action, LanguageVersion greaterOrEqualThanLanguageVersion)
        {
            if (languageVersion >= greaterOrEqualThanLanguageVersion)
action?.Invoke();
        }

        public static void
RegisterSyntaxNodeActionForVersionLower<TLanguageKindEnum>(this AnalysisContext
context, LanguageVersion lowerThanLanguageVersion,
        Action<SyntaxNodeAnalysisContext> action, params TLanguageKindEnum[]
syntaxKinds) where TLanguageKindEnum : struct =>

context.RegisterCompilationStartActionForVersionLower(lowerThanLanguageVersion,
compilationContext => compilationContext.RegisterSyntaxNodeAction(action,
syntaxKinds));
```

```
        public static void RegisterCompilationStartActionForVersionLower(this
AnalysisContext context, LanguageVersion lowerThanLanguageVersion,
Action<CompilationStartAnalysisContext> registrationAction) =>
            context.RegisterCompilationStartAction(compilationContext =>
compilationContext.RunIfCSharpVersionLower(lowerThanLanguageVersion, () =>
registrationAction?.Invoke(compilationContext)));
#pragma warning disable RS1012
        private static void RunIfCSharpVersionLower(this
CompilationStartAnalysisContext context, LanguageVersion lowerThanLanguageVersion,
Action action) =>
            context.Compilation.RunIfCSharpVersionLower(action,
lowerThanLanguageVersion);
#pragma warning restore RS1012
        private static void RunIfCSharpVersionLower(this Compilation compilation,
Action action, LanguageVersion lowerThanLanguageVersion) =>
            (compilation as
CSharpCompilation)?.LanguageVersion.RunIfCSharpVersionLower(action,
lowerThanLanguageVersion);

        private static void RunIfCSharpVersionLower(this LanguageVersion
languageVersion, Action action, LanguageVersion lowerThanLanguageVersion)
        {
            if (languageVersion < lowerThanLanguageVersion) action?.Invoke();
        }

        public static ConditionalAccessExpressionSyntax
ToConditionalAccessExpression(this MemberAccessExpressionSyntax memberAccess) =>
            SyntaxFactory.ConditionalAccessExpression(memberAccess.Expression,
SyntaxFactory.MemberBindingExpression(memberAccess.Name));

        public static StatementSyntax GetSingleStatementFromPossibleBlock(this
StatementSyntax statement)
        {
            var block = statement as BlockSyntax;
            if (block != null)
            {
                if (block.Statements.Count != 1) return null;
                return block.Statements.Single();
            }
            else
            {
                return statement;
            }
        }

        public static bool IsEmbeddedStatementOwner(this SyntaxNode node)
        {
            return node is IfStatementSyntax ||
                   node is ElseClauseSyntax ||
                   node is ForStatementSyntax ||
                   node is ForEachStatementSyntax ||
                   node is WhileStatementSyntax ||
                   node is UsingStatementSyntax ||
                   node is DoStatementSyntax ||
                   node is LockStatementSyntax ||
                   node is FixedStatementSyntax;
        }

        public static IEnumerable<TypeDeclarationSyntax> DescendantTypes(this
SyntaxNode root)
```

```
        {
            return root
                .DescendantNodes(n => !(n.IsKind(
                    SyntaxKind.MethodDeclaration,
                    SyntaxKind.ConstructorDeclaration,
                    SyntaxKind.DelegateDeclaration,
                    SyntaxKind.DestructorDeclaration,
                    SyntaxKind.EnumDeclaration,
                    SyntaxKind.PropertyDeclaration,
                    SyntaxKind.FieldDeclaration,
                    SyntaxKind.InterfaceDeclaration,
                    SyntaxKind.PropertyDeclaration,
                    SyntaxKind.EventDeclaration)))
                .OfType<TypeDeclarationSyntax>();
        }

        public static T GetAncestor<T>(this SyntaxToken token, Func<T, bool>
predicate = null)
            where T : SyntaxNode => token.Parent?.FirstAncestorOrSelf(predicate);

        public static bool IsKind(this SyntaxToken token, params SyntaxKind[] kinds)
        {
            foreach (var kind in kinds)
                if (Microsoft.CodeAnalysis.CSharpExtensions.IsKind(token, kind))
return true;
            return false;
        }

        public static bool IsKind(this SyntaxTrivia trivia, params SyntaxKind[]
kinds)
        {
            foreach (var kind in kinds)
                if (Microsoft.CodeAnalysis.CSharpExtensions.IsKind(trivia, kind))
return true;
            return false;
        }

        public static bool IsKind(this SyntaxNode node, params SyntaxKind[] kinds)
        {
            foreach (var kind in kinds)
                if (Microsoft.CodeAnalysis.CSharpExtensions.IsKind(node, kind))
return true;
            return false;
        }

        public static bool IsKind(this SyntaxNodeOrToken nodeOrToken, params
SyntaxKind[] kinds)
        {
            foreach (var kind in kinds)
                if (Microsoft.CodeAnalysis.CSharpExtensions.IsKind(nodeOrToken,
kind)) return true;
            return false;
        }

        public static bool IsNotKind(this SyntaxNode node, params SyntaxKind[]
kinds) => !node.IsKind(kinds);

        public static bool Any(this SyntaxTokenList list, SyntaxKind kind1,
SyntaxKind kind2) =>
            list.IndexOf(kind1) >= 0 || list.IndexOf(kind2) >= 0;
```

```
        public static bool Any(this SyntaxTokenList list, SyntaxKind kind1,
SyntaxKind kind2, params SyntaxKind[] kinds)
        {
            if (list.Any(kind1, kind2)) return true;
            for (int i = 0; i < kinds.Length; i++)
                if (list.IndexOf(kinds[i]) >= 0) return true;
            return false;
        }

        public static bool IsAnyKind(this SyntaxNode node, params SyntaxKind[]
kinds)
        {
            foreach (var kind in kinds)
            {
                if (node.IsKind(kind)) return true;
            }
            return false;
        }

        public static MemberDeclarationSyntax FirstAncestorOrSelfThatIsAMember(this
SyntaxNode node)
        {
            var currentNode = node;
            while (true)
            {
                if (currentNode == null) break;
                if (currentNode.IsAnyKind(
                    SyntaxKind.EnumDeclaration, SyntaxKind.ClassDeclaration,
                    SyntaxKind.InterfaceDeclaration, SyntaxKind.StructDeclaration,
                    SyntaxKind.ConstructorDeclaration,
SyntaxKind.DestructorDeclaration,
                    SyntaxKind.MethodDeclaration, SyntaxKind.PropertyDeclaration,
                    SyntaxKind.EventDeclaration, SyntaxKind.DelegateDeclaration,
                    SyntaxKind.EventFieldDeclaration, SyntaxKind.FieldDeclaration,
                    SyntaxKind.ConversionOperatorDeclaration,
SyntaxKind.OperatorDeclaration,
                    SyntaxKind.IndexerDeclaration, SyntaxKind.NamespaceDeclaration))
                    return (MemberDeclarationSyntax)currentNode;
                currentNode = currentNode.Parent;
            }
            return null;

        }

        public static StatementSyntax FirstAncestorOrSelfThatIsAStatement(this
SyntaxNode node)
        {
            var currentNode = node;
            while (true)
            {
                if (currentNode == null) break;
                if (currentNode.IsAnyKind(SyntaxKind.Block,
SyntaxKind.BreakStatement,
                    SyntaxKind.CheckedStatement, SyntaxKind.ContinueStatement,
                    SyntaxKind.DoStatement, SyntaxKind.EmptyStatement,
                    SyntaxKind.ExpressionStatement, SyntaxKind.FixedKeyword,
                    SyntaxKind.ForEachKeyword, SyntaxKind.ForStatement,
                    SyntaxKind.GotoStatement, SyntaxKind.IfStatement,
                    SyntaxKind.LabeledStatement,
SyntaxKind.LocalDeclarationStatement,
                    SyntaxKind.LockStatement, SyntaxKind.ReturnStatement,
```

```
                    SyntaxKind.SwitchStatement, SyntaxKind.ThrowStatement,
                    SyntaxKind.TryStatement, SyntaxKind.UnsafeStatement,
                    SyntaxKind.UsingStatement, SyntaxKind.WhileStatement,
                    SyntaxKind.YieldBreakStatement,
SyntaxKind.YieldReturnStatement))
                    return (StatementSyntax)currentNode;
                currentNode = currentNode.Parent;
            }
            return null;
        }

        public static bool HasAttributeOnAncestorOrSelf(this SyntaxNode node, string
attributeName)
        {
            var csharpNode = node as CSharpSyntaxNode;
            if (csharpNode == null) throw new Exception("Node is not a C# node");
            return csharpNode.HasAttributeOnAncestorOrSelf(attributeName);
        }

        public static bool HasAttributeOnAncestorOrSelf(this SyntaxNode node, params
string[] attributeNames)
        {
            var csharpNode = node as CSharpSyntaxNode;
            if (csharpNode == null) throw new Exception("Node is not a C# node");
            foreach (var attributeName in attributeNames)
                if (csharpNode.HasAttributeOnAncestorOrSelf(attributeName)) return
true;
            return false;
        }

        public static bool HasAttributeOnAncestorOrSelf(this CSharpSyntaxNode node,
string attributeName)
        {
            var parentMethod =
(BaseMethodDeclarationSyntax)node.FirstAncestorOrSelfOfType(typeof(MethodDeclaration
Syntax), typeof(ConstructorDeclarationSyntax));
            if (parentMethod?.AttributeLists.HasAttribute(attributeName) ?? false)
                return true;
            var type =
(TypeDeclarationSyntax)node.FirstAncestorOrSelfOfType(typeof(ClassDeclarationSyntax)
, typeof(StructDeclarationSyntax));
            while (type != null)
            {
                if (type.AttributeLists.HasAttribute(attributeName))
                    return true;
                type =
(TypeDeclarationSyntax)type.FirstAncestorOfType(typeof(ClassDeclarationSyntax),
typeof(StructDeclarationSyntax));
            }
            var property =
node.FirstAncestorOrSelfOfType<PropertyDeclarationSyntax>();
            if (property?.AttributeLists.HasAttribute(attributeName) ?? false)
                return true;
            var accessor =
node.FirstAncestorOrSelfOfType<AccessorDeclarationSyntax>();
            if (accessor?.AttributeLists.HasAttribute(attributeName) ?? false)
                return true;
            var anInterface =
node.FirstAncestorOrSelfOfType<InterfaceDeclarationSyntax>();
            if (anInterface?.AttributeLists.HasAttribute(attributeName) ?? false)
                return true;
```

```
            var anEvent = node.FirstAncestorOrSelfOfType<EventDeclarationSyntax>();
            if (anEvent?.AttributeLists.HasAttribute(attributeName) ?? false)
                return true;
            var anEnum = node.FirstAncestorOrSelfOfType<EnumDeclarationSyntax>();
            if (anEnum?.AttributeLists.HasAttribute(attributeName) ?? false)
                return true;
            var field = node.FirstAncestorOrSelfOfType<FieldDeclarationSyntax>();
            if (field?.AttributeLists.HasAttribute(attributeName) ?? false)
                return true;
            var eventField =
node.FirstAncestorOrSelfOfType<EventFieldDeclarationSyntax>();
            if (eventField?.AttributeLists.HasAttribute(attributeName) ?? false)
                return true;
            var parameter = node as ParameterSyntax;
            if (parameter?.AttributeLists.HasAttribute(attributeName) ?? false)
                return true;
            var aDelegate = node as DelegateDeclarationSyntax;
            if (aDelegate?.AttributeLists.HasAttribute(attributeName) ?? false)
                return true;
            return false;
        }

        public static bool HasAttribute(this SyntaxList<AttributeListSyntax>
attributeLists, string attributeName) =>
            attributeLists.SelectMany(a => a.Attributes).Any(a =>
a.Name.ToString().EndsWith(attributeName, StringComparison.OrdinalIgnoreCase));

        public static bool HasAnyAttribute(this SyntaxList<AttributeListSyntax>
attributeLists, string[] attributeNames) =>
            attributeLists.SelectMany(a => a.Attributes).Select(a =>
a.Name.ToString()).Any(name => attributeNames.Any(attributeName =>
            name.EndsWith(attributeName, StringComparison.OrdinalIgnoreCase)
            || name.EndsWith($"{attributeName}Attribute",
StringComparison.OrdinalIgnoreCase)));

        public static NameSyntax ToNameSyntax(this INamespaceSymbol namespaceSymbol)
=>
            ToNameSyntax(namespaceSymbol.ToDisplayString().Split('.'));

        private static NameSyntax ToNameSyntax(IEnumerable<string> names)
        {
            var count = names.Count();
            if (count == 1)
                return SyntaxFactory.IdentifierName(names.First());
            return SyntaxFactory.QualifiedName(
                ToNameSyntax(names.Take(count - 1)),
                ToNameSyntax(names.Skip(count - 1)) as IdentifierNameSyntax
            );
        }

        public static TypeSyntax FindTypeInParametersList(this
SeparatedSyntaxList<ParameterSyntax> parameterList, string typeName)
        {
            TypeSyntax result = null;
            var lastIdentifierOfTypeName =
typeName.GetLastIdentifierIfQualiedTypeName();
            foreach (var parameter in parameterList)
            {
                var valueText = GetLastIdentifierValueText(parameter.Type);

                if (!string.IsNullOrEmpty(valueText))
```

```
                {
                    if (string.Equals(valueText, lastIdentifierOfTypeName,
StringComparison.Ordinal))
                    {
                        result = parameter.Type;
                        break;
                    }
                }
            }

            return result;
        }

        private static string GetLastIdentifierValueText(CSharpSyntaxNode node)
        {
            var result = string.Empty;
            switch (node.Kind())
            {
                case SyntaxKind.IdentifierName:
                    result = ((IdentifierNameSyntax)node).Identifier.ValueText;
                    break;
                case SyntaxKind.QualifiedName:
                    result =
GetLastIdentifierValueText(((QualifiedNameSyntax)node).Right);
                    break;
                case SyntaxKind.GenericName:
                    var genericNameSyntax = ((GenericNameSyntax)node);
                    result =
$"{genericNameSyntax.Identifier.ValueText}{genericNameSyntax.TypeArgumentList.ToStri
ng()}";
                    break;
                case SyntaxKind.AliasQualifiedName:
                    result =
((AliasQualifiedNameSyntax)node).Name.Identifier.ValueText;
                    break;
                default:
                    break;
            }
            return result;
        }

        public static SyntaxToken GetIdentifier(this BaseMethodDeclarationSyntax
method)
        {
            var result = default(SyntaxToken);

            switch (method.Kind())
            {
                case SyntaxKind.MethodDeclaration:
                    result = ((MethodDeclarationSyntax)method).Identifier;
                    break;
                case SyntaxKind.ConstructorDeclaration:
                    result = ((ConstructorDeclarationSyntax)method).Identifier;
                    break;
                case SyntaxKind.DestructorDeclaration:
                    result = ((DestructorDeclarationSyntax)method).Identifier;
                    break;
                default:
                    return result;
            }
            return result;
```

```
        }

        public static MemberDeclarationSyntax WithModifiers(this
MemberDeclarationSyntax declaration, SyntaxTokenList newModifiers)
        {
            var result = declaration;

            switch (declaration.Kind())
            {
                case SyntaxKind.ClassDeclaration:
                    result =
((ClassDeclarationSyntax)declaration).WithModifiers(newModifiers);
                    break;
                case SyntaxKind.StructDeclaration:
                    result =
((StructDeclarationSyntax)declaration).WithModifiers(newModifiers);
                    break;
                case SyntaxKind.InterfaceDeclaration:
                    result =
((InterfaceDeclarationSyntax)declaration).WithModifiers(newModifiers);
                    break;
                case SyntaxKind.EnumDeclaration:
                    result =
((EnumDeclarationSyntax)declaration).WithModifiers(newModifiers);
                    break;
                case SyntaxKind.DelegateDeclaration:
                    result =
((DelegateDeclarationSyntax)declaration).WithModifiers(newModifiers);
                    break;
                case SyntaxKind.FieldDeclaration:
                    result =
((FieldDeclarationSyntax)declaration).WithModifiers(newModifiers);
                    break;
                case SyntaxKind.EventFieldDeclaration:
                    result =
((EventFieldDeclarationSyntax)declaration).WithModifiers(newModifiers);
                    break;
                case SyntaxKind.MethodDeclaration:
                    result =
((MethodDeclarationSyntax)declaration).WithModifiers(newModifiers);
                    break;
                case SyntaxKind.OperatorDeclaration:
                    result =
((OperatorDeclarationSyntax)declaration).WithModifiers(newModifiers);
                    break;
                case SyntaxKind.ConversionOperatorDeclaration:
                    result =
((ConversionOperatorDeclarationSyntax)declaration).WithModifiers(newModifiers);
                    break;
                case SyntaxKind.ConstructorDeclaration:
                    result =
((ConstructorDeclarationSyntax)declaration).WithModifiers(newModifiers);
                    break;
                case SyntaxKind.DestructorDeclaration:
                    result =
((DestructorDeclarationSyntax)declaration).WithModifiers(newModifiers);
                    break;
                case SyntaxKind.PropertyDeclaration:
                    result =
((PropertyDeclarationSyntax)declaration).WithModifiers(newModifiers);
                    break;
```

```
                case SyntaxKind.IndexerDeclaration:
                    result =
((IndexerDeclarationSyntax)declaration).WithModifiers(newModifiers);
                    break;
                case SyntaxKind.EventDeclaration:
                    result =
((EventDeclarationSyntax)declaration).WithModifiers(newModifiers);
                    break;
                default:
                    break;
            }

            return result;
        }

        public static SyntaxTokenList GetModifiers(this MemberDeclarationSyntax
memberDeclaration)
        {
            var result = default(SyntaxTokenList);

            switch (memberDeclaration.Kind())
            {
                case SyntaxKind.ClassDeclaration:
                case SyntaxKind.StructDeclaration:
                case SyntaxKind.InterfaceDeclaration:
                case SyntaxKind.EnumDeclaration:
                    result =
((BaseTypeDeclarationSyntax)memberDeclaration).Modifiers;
                    break;
                case SyntaxKind.DelegateDeclaration:
                    result =
((DelegateDeclarationSyntax)memberDeclaration).Modifiers;
                    break;
                case SyntaxKind.FieldDeclaration:
                case SyntaxKind.EventFieldDeclaration:
                    result =
((BaseFieldDeclarationSyntax)memberDeclaration).Modifiers;
                    break;
                case SyntaxKind.MethodDeclaration:
                case SyntaxKind.OperatorDeclaration:
                case SyntaxKind.ConversionOperatorDeclaration:
                case SyntaxKind.ConstructorDeclaration:
                case SyntaxKind.DestructorDeclaration:
                    result =
((BaseMethodDeclarationSyntax)memberDeclaration).Modifiers;
                    break;
                case SyntaxKind.PropertyDeclaration:
                case SyntaxKind.IndexerDeclaration:
                case SyntaxKind.EventDeclaration:
                    result =
((BasePropertyDeclarationSyntax)memberDeclaration).Modifiers;
                    break;
                default:
                    break;
            }

            return result;
        }

        public static SyntaxTokenList CloneAccessibilityModifiers(this
BaseMethodDeclarationSyntax method)
```

```
        {
            var modifiers = method.Modifiers;
            if (method.Parent.IsKind(SyntaxKind.InterfaceDeclaration))
            {
                modifiers = ((InterfaceDeclarationSyntax)method.Parent).Modifiers;
            }

            return modifiers.CloneAccessibilityModifiers();
        }

        public static SyntaxTokenList CloneAccessibilityModifiers(this
SyntaxTokenList modifiers)
        {
            var accessibilityModifiers = modifiers.Where(token =>
token.IsKind(SyntaxKind.PublicKeyword) || token.IsKind(SyntaxKind.ProtectedKeyword)
|| token.IsKind(SyntaxKind.InternalKeyword) ||
token.IsKind(SyntaxKind.PrivateKeyword)).Select(token =>
SyntaxFactory.Token(token.Kind()));

            return
SyntaxFactory.TokenList(accessibilityModifiers.EnsureProtectedBeforeInternal());
        }

        public static SyntaxNode FirstAncestorOfKind(this SyntaxNode node, params
SyntaxKind[] kinds)
        {
            var currentNode = node;
            while (true)
            {
                var parent = currentNode.Parent;
                if (parent == null) break;
                if (parent.IsAnyKind(kinds)) return parent;
                currentNode = parent;
            }
            return null;
        }

        public static TNode FirstAncestorOfKind<TNode>(this SyntaxNode node, params
SyntaxKind[] kinds) where TNode : SyntaxNode =>
            (TNode)FirstAncestorOfKind(node, kinds);

        public static IEnumerable<TNode> OfKind<TNode>(this IEnumerable<SyntaxNode>
nodes, SyntaxKind kind) where TNode : SyntaxNode
        {
            foreach (var node in nodes)
                if (node.IsKind(kind))
                    yield return (TNode)node;
        }

        public static IEnumerable<TNode> OfKind<TNode>(this IEnumerable<SyntaxNode>
nodes, params SyntaxKind[] kinds) where TNode : SyntaxNode
        {
            foreach (var node in nodes)
                if (node.IsAnyKind(kinds))
                    yield return (TNode)node;
        }

        public static IEnumerable<TNode> OfKind<TNode>(this IEnumerable<TNode>
nodes, SyntaxKind kind) where TNode : SyntaxNode
        {
            foreach (var node in nodes)
```

```
                if (node.IsKind(kind))
                    yield return node;
        }

        public static IEnumerable<TNode> OfKind<TNode>(this IEnumerable<TNode>
nodes, params SyntaxKind[] kinds) where TNode : SyntaxNode
        {
            foreach (var node in nodes)
                if (node.IsAnyKind(kinds))
                    yield return node;
        }

        public static StatementSyntax GetPreviousStatement(this StatementSyntax
statement)
        {
            var parent = statement.Parent;
            SyntaxList<StatementSyntax> statements;
            if (parent.IsKind(SyntaxKind.Block))
            {
                var block = (BlockSyntax)parent;
                statements = block.Statements;
            }
            else if (parent.IsKind(SyntaxKind.SwitchSection))
            {
                var section = (SwitchSectionSyntax)parent;
                statements = section.Statements;
            }
            else return null;
            if (statement.Equals(statements[0])) return null;
            for (int i = 1; i < statements.Count; i++)
            {
                var someStatement = statements[i];
                if (statement.Equals(someStatement))
                    return statements[i - 1];
            }
            return null;
        }


        /// <summary>
        /// Determines whether the specified symbol is a read only field and
initialized in the specified context.
        /// </summary>
        /// <param name="context">The context.</param>
        /// <param name="symbol">The symbol.</param>
        /// <returns>
        /// True if the symbol is a read only field that is initialized either on
declaration or in all constructors of the containing type; otherwise false.
        /// </returns>
        /// <remarks>
        /// If the symbol is initialized in a block of code of the constructor that
might not always be called, the symbol is considered to
        /// not be initialized for certain. For more information <seealso
cref="DoesBlockContainDefiniteInitializer(SyntaxNodeAnalysisContext, ISymbol,
IEnumerable{StatementSyntax})"/>
        /// </remarks>
        public static bool IsReadOnlyAndInitializedForCertain(this ISymbol symbol,
SyntaxNodeAnalysisContext context)
        {
            if (symbol.Kind != SymbolKind.Field) return false;
```

```
            var field = (IFieldSymbol)symbol;
            foreach (var declaringSyntaxReference in
symbol.DeclaringSyntaxReferences)
            {
                var variableDeclarator =
declaringSyntaxReference.GetSyntax(context.CancellationToken) as
VariableDeclaratorSyntax;

                if (variableDeclarator != null && variableDeclarator.Initializer !=
null && field.IsReadOnly &&

!variableDeclarator.Initializer.Value.IsKind(SyntaxKind.NullLiteralExpression))
return true;
            }

            foreach (var constructor in symbol.ContainingType.Constructors)
            {
                foreach (var declaringSyntaxReference in
constructor.DeclaringSyntaxReferences)
                {
                    var constructorSyntax =
declaringSyntaxReference.GetSyntax(context.CancellationToken) as
ConstructorDeclarationSyntax;
                    if (constructorSyntax != null)
                        if (field.IsReadOnly &&
constructorSyntax.Body.Statements.DoesBlockContainCertainInitializer(context,
symbol) == InitializerState.Initializer)
                            return true;
                }
            }

            return false;
        }

        private static InitializerState DoesBlockContainCertainInitializer(this
StatementSyntax statement, SyntaxNodeAnalysisContext context, ISymbol symbol)
        {
            return new[] { statement }.DoesBlockContainCertainInitializer(context,
symbol);
        }

        /// <summary>
        /// This method can be used to determine if the specified block of
        /// statements contains an initializer for the specified symbol.
        /// </summary>
        /// <param name="context">The context.</param>
        /// <param name="symbol">The symbol.</param>
        /// <param name="statements">The statements.</param>
        /// <returns>
        /// The initializer state found
        /// </returns>
        /// <remarks>
        /// Code blocks that might not always be called are:
        /// - An if or else statement.
        /// - The body of a for, while or for-each loop.
        /// - Switch statements
        ///
        /// The following exceptions are taken into account:
        /// - If both if and else statements contain a certain initialization.
        /// - If all cases in a switch contain a certain initialization (this means
a default case must exist as well).
```

```
        ///
        /// Please note that this is a recursive function so we can check a block of
code in an if statement for example.
        /// </remarks>
        private static InitializerState DoesBlockContainCertainInitializer(this
IEnumerable<StatementSyntax> statements, SyntaxNodeAnalysisContext context, ISymbol
symbol)
        {
            // Keep track of the current initializer state. This can only be None
            // or Initializer, WayToSkipInitializer will always be returned
immediately.
            // Only way to go back from Initializer to None is if there is an
assignment
            // to null after a previous assignment to a non-null value.
            var currentState = InitializerState.None;

            foreach (var statement in statements)
            {
                if (statement.IsKind(SyntaxKind.ReturnStatement) && currentState ==
InitializerState.None)
                {
                    return InitializerState.WayToSkipInitializer;
                }
                else if (statement.IsKind(SyntaxKind.Block))
                {
                    var blockResult =
((BlockSyntax)statement).Statements.DoesBlockContainCertainInitializer(context,
symbol);
                    if (CanSkipInitializer(blockResult, currentState))
                        return InitializerState.WayToSkipInitializer;
                    if (blockResult == InitializerState.Initializer)
                        currentState = blockResult;
                }
                else if (statement.IsKind(SyntaxKind.UsingStatement))
                {
                    var blockResult =
((UsingStatementSyntax)statement).Statement.DoesBlockContainCertainInitializer(conte
xt, symbol);
                    if (CanSkipInitializer(blockResult, currentState))
                        return InitializerState.WayToSkipInitializer;
                    if (blockResult == InitializerState.Initializer)
                        currentState = blockResult;
                }
                else if (statement.IsKind(SyntaxKind.ExpressionStatement))
                {
                    var expression =
((ExpressionStatementSyntax)statement).Expression;
                    if (expression.IsKind(SyntaxKind.SimpleAssignmentExpression))
                    {
                        var assignmentExpression =
(AssignmentExpressionSyntax)expression;
                        var identifier = assignmentExpression.Left;
                        if (identifier != null)
                        {
                            var right = assignmentExpression.Right;
                            if (right != null)
                            {
                                if (right.IsKind(SyntaxKind.NullLiteralExpression))
                                    currentState = InitializerState.None;
                                else if
(symbol.Equals(context.SemanticModel.GetSymbolInfo(identifier).Symbol))
```

```
                                    currentState = InitializerState.Initializer;
                                }
                            }
                        }
                    }
                    else if (statement.IsKind(SyntaxKind.SwitchStatement))
                    {
                        var switchStatement = (SwitchStatementSyntax)statement;
                        if (switchStatement.Sections.Any(s => s.Labels.Any(l =>
l.IsKind(SyntaxKind.DefaultSwitchLabel))))
                        {
                            var sectionInitializerStates =
switchStatement.Sections.Select(s =>
s.Statements.DoesBlockContainCertainInitializer(context, symbol)).ToList();
                            if (sectionInitializerStates.All(sectionInitializerState =>
sectionInitializerState == InitializerState.Initializer))
                                currentState = InitializerState.Initializer;
                            else if
(sectionInitializerStates.Any(sectionInitializerState =>
CanSkipInitializer(sectionInitializerState, currentState)))
                                return InitializerState.WayToSkipInitializer;
                        }
                    }
                    else if (statement.IsKind(SyntaxKind.IfStatement))
                    {
                        var ifStatement = (IfStatementSyntax)statement;

                        var ifResult =
ifStatement.Statement.DoesBlockContainCertainInitializer(context, symbol);
                        if (ifStatement.Else != null)
                        {
                            var elseResult =
ifStatement.Else.Statement.DoesBlockContainCertainInitializer(context, symbol);

                            if (ifResult == InitializerState.Initializer && elseResult
== InitializerState.Initializer)
                                currentState = InitializerState.Initializer;
                            if (CanSkipInitializer(elseResult, currentState))
                                return InitializerState.WayToSkipInitializer;
                        }
                        if (CanSkipInitializer(ifResult, currentState))
                        {
                            return InitializerState.WayToSkipInitializer;
                        }
                    }
                }
                return currentState;
            }

            private static bool CanSkipInitializer(InitializerState foundState,
InitializerState currentState) =>
                foundState == InitializerState.WayToSkipInitializer && currentState ==
InitializerState.None;

            public static TNode WithoutAllTrivia<TNode>(this TNode node) where TNode :
SyntaxNode
            {
                var newNode = node.WithoutTrivia();
                var tokens = newNode.ChildTokens().ToList();
                var newTokens = tokens.ToDictionary(t => t, t => t.WithoutTrivia());
                newNode = newNode.ReplaceTokens(tokens, (o, _) => newTokens[o]);
```

```
            var nodes = newNode.ChildNodes().ToList();
            var newNodes = nodes.ToDictionary(n => n, n => n.WithoutAllTrivia());
            newNode = newNode.ReplaceNodes(nodes, (o, _) => newNodes[o]);
            newNode = newNode.WithAdditionalAnnotations(Formatter.Annotation);
            return newNode;
        }

        public static SyntaxToken WithoutTrivia(this SyntaxToken token)
        {
            var trivia = token.GetAllTrivia();
            var newToken = token.ReplaceTrivia(trivia, (o, _) =>
default(SyntaxTrivia));
            return newToken;
        }

        private static readonly SyntaxTokenList publicToken =
SyntaxFactory.TokenList(SyntaxFactory.Token(SyntaxKind.PublicKeyword));
        private static readonly SyntaxTokenList privateToken =
SyntaxFactory.TokenList(SyntaxFactory.Token(SyntaxKind.PrivateKeyword));
        private static readonly SyntaxTokenList protectedToken =
SyntaxFactory.TokenList(SyntaxFactory.Token(SyntaxKind.ProtectedKeyword));
        private static readonly SyntaxTokenList protectedInternalToken =
SyntaxFactory.TokenList(
            SyntaxFactory.Token(SyntaxKind.ProtectedKeyword),
SyntaxFactory.Token(SyntaxKind.InternalKeyword));
        private static readonly SyntaxTokenList internalToken =
SyntaxFactory.TokenList(SyntaxFactory.Token(SyntaxKind.InternalKeyword));
        public static SyntaxTokenList GetTokens(this Accessibility accessibility)
        {
            switch (accessibility)
            {
                case Accessibility.Public:
                    return publicToken;
                case Accessibility.Private:
                    return privateToken;
                case Accessibility.Protected:
                    return protectedToken;
                case Accessibility.Internal:
                    return internalToken;
                case Accessibility.ProtectedAndInternal:
                    return protectedInternalToken;
                default:
                    throw new NotSupportedException();
            }
        }
        public static TypeDeclarationSyntax WithMembers(this TypeDeclarationSyntax
typeDeclarationSyntax, SyntaxList<MemberDeclarationSyntax> members)
        {
            if (typeDeclarationSyntax is ClassDeclarationSyntax)
            {
                return (typeDeclarationSyntax as
ClassDeclarationSyntax).WithMembers(members);
            }
            else if (typeDeclarationSyntax is StructDeclarationSyntax)
            {
                return (typeDeclarationSyntax as
StructDeclarationSyntax).WithMembers(members);
            }
            else
            {
                throw new NotSupportedException();
```

```
            }
        }

        /// <summary>
        /// According to the C# Language Spec, item 6.4
        /// See <a
href="https://github.com/ljw1004/csharpspec/blob/master/csharp/conversions.md#implic
it-numeric-conversions">online</a>.
        /// </summary>
        /// <param name="from">The type to convert from</param>
        /// <param name="to">The type to convert to</param>
        public static bool HasImplicitNumericConversion(this ITypeSymbol from,
ITypeSymbol to)
        {
            if (from == null || to == null) return false;
            switch (from.SpecialType)
            {
                case SpecialType.System_SByte:
                    switch (to.SpecialType)
                    {
                        case SpecialType.System_SByte:
                        case SpecialType.System_Int16:
                        case SpecialType.System_Int32:
                        case SpecialType.System_Int64:
                        case SpecialType.System_Single:
                        case SpecialType.System_Double:
                        case SpecialType.System_Decimal:
                            return true;
                        default:
                            return false;
                    }
                case SpecialType.System_Byte:
                    switch (to.SpecialType)
                    {
                        case SpecialType.System_Byte:
                        case SpecialType.System_Int16:
                        case SpecialType.System_UInt16:
                        case SpecialType.System_Int32:
                        case SpecialType.System_UInt32:
                        case SpecialType.System_Int64:
                        case SpecialType.System_UInt64:
                        case SpecialType.System_Single:
                        case SpecialType.System_Double:
                        case SpecialType.System_Decimal:
                            return true;
                        default:
                            return false;
                    }
                case SpecialType.System_Int16:
                    switch (to.SpecialType)
                    {
                        case SpecialType.System_Int16:
                        case SpecialType.System_Int32:
                        case SpecialType.System_Int64:
                        case SpecialType.System_Single:
                        case SpecialType.System_Double:
                        case SpecialType.System_Decimal:
                            return true;
                        default:
                            return false;
                    }
```

```
case SpecialType.System_UInt16:
    switch (to.SpecialType)
    {
        case SpecialType.System_Int32:
        case SpecialType.System_UInt32:
        case SpecialType.System_Int64:
        case SpecialType.System_UInt64:
        case SpecialType.System_Single:
        case SpecialType.System_Double:
        case SpecialType.System_Decimal:
            return true;
        default:
            return false;
    }
case SpecialType.System_Int32:
    switch (to.SpecialType)
    {
        case SpecialType.System_Int32:
        case SpecialType.System_Int64:
        case SpecialType.System_Single:
        case SpecialType.System_Double:
        case SpecialType.System_Decimal:
            return true;
        default:
            return false;
    }
case SpecialType.System_UInt32:
    switch (to.SpecialType)
    {
        case SpecialType.System_UInt32:
        case SpecialType.System_Int64:
        case SpecialType.System_UInt64:
        case SpecialType.System_Single:
        case SpecialType.System_Double:
        case SpecialType.System_Decimal:
            return true;
        default:
            return false;
    }
case SpecialType.System_Int64:
    switch (to.SpecialType)
    {
        case SpecialType.System_Int64:
        case SpecialType.System_Single:
        case SpecialType.System_Double:
        case SpecialType.System_Decimal:
            return true;
        default:
            return false;
    }
case SpecialType.System_UInt64:
    switch (to.SpecialType)
    {
        case SpecialType.System_UInt64:
        case SpecialType.System_Single:
        case SpecialType.System_Double:
        case SpecialType.System_Decimal:
            return true;
        default:
            return false;
    }
```

```
                case SpecialType.System_Char:
                    switch (to.SpecialType)
                    {
                        case SpecialType.System_UInt16:
                        case SpecialType.System_Int32:
                        case SpecialType.System_UInt32:
                        case SpecialType.System_Int64:
                        case SpecialType.System_UInt64:
                        case SpecialType.System_Char:
                        case SpecialType.System_Single:
                        case SpecialType.System_Double:
                        case SpecialType.System_Decimal:
                            return true;
                        default:
                            return false;
                    }
                case SpecialType.System_Single:
                    switch (to.SpecialType)
                    {
                        case SpecialType.System_Single:
                        case SpecialType.System_Double:
                            return true;
                        default:
                            return false;
                    }
                default:
                    return false;
            }
        }

        public static string FindAvailableIdentifierName(this SemanticModel
semanticModel, int position, string baseName)
        {
            var name = baseName;
            var inscrementer = 1;
            while (semanticModel.LookupSymbols(position, name: name).Any())
                name = baseName + inscrementer++;
            return name;
        }

        public static bool IsImplementingInterface(this MemberDeclarationSyntax
member, SemanticModel semanticModel) =>
            semanticModel.GetDeclaredSymbol(member).IsImplementingInterface();

        public static bool IsImplementingInterface(this ISymbol memberSymbol)
        {
            if (memberSymbol == null) return false;
            IMethodSymbol methodSymbol;
            IEventSymbol eventSymbol;
            IPropertySymbol propertySymbol;
            if ((methodSymbol = memberSymbol as IMethodSymbol) != null)
            {
                if (methodSymbol.ExplicitInterfaceImplementations.Any()) return
true;
            }
            else if ((propertySymbol = memberSymbol as IPropertySymbol) != null)
            {
                if (propertySymbol.ExplicitInterfaceImplementations.Any()) return
true;
            }
            else if ((eventSymbol = memberSymbol as IEventSymbol) != null)
```

```
            {
                if (eventSymbol.ExplicitInterfaceImplementations.Any()) return true;
            }
            else return false;
            var type = memberSymbol.ContainingType;
            if (type == null) return false;
            var interfaceMembersWithSameName = type.AllInterfaces.SelectMany(i =>
i.GetMembers(memberSymbol.Name));
            foreach (var interfaceMember in interfaceMembersWithSameName)
            {
                var implementation =
type.FindImplementationForInterfaceMember(interfaceMember);
                if (implementation != null && implementation.Equals(memberSymbol))
return true;
            }
            return false;
        }
    }
}
```

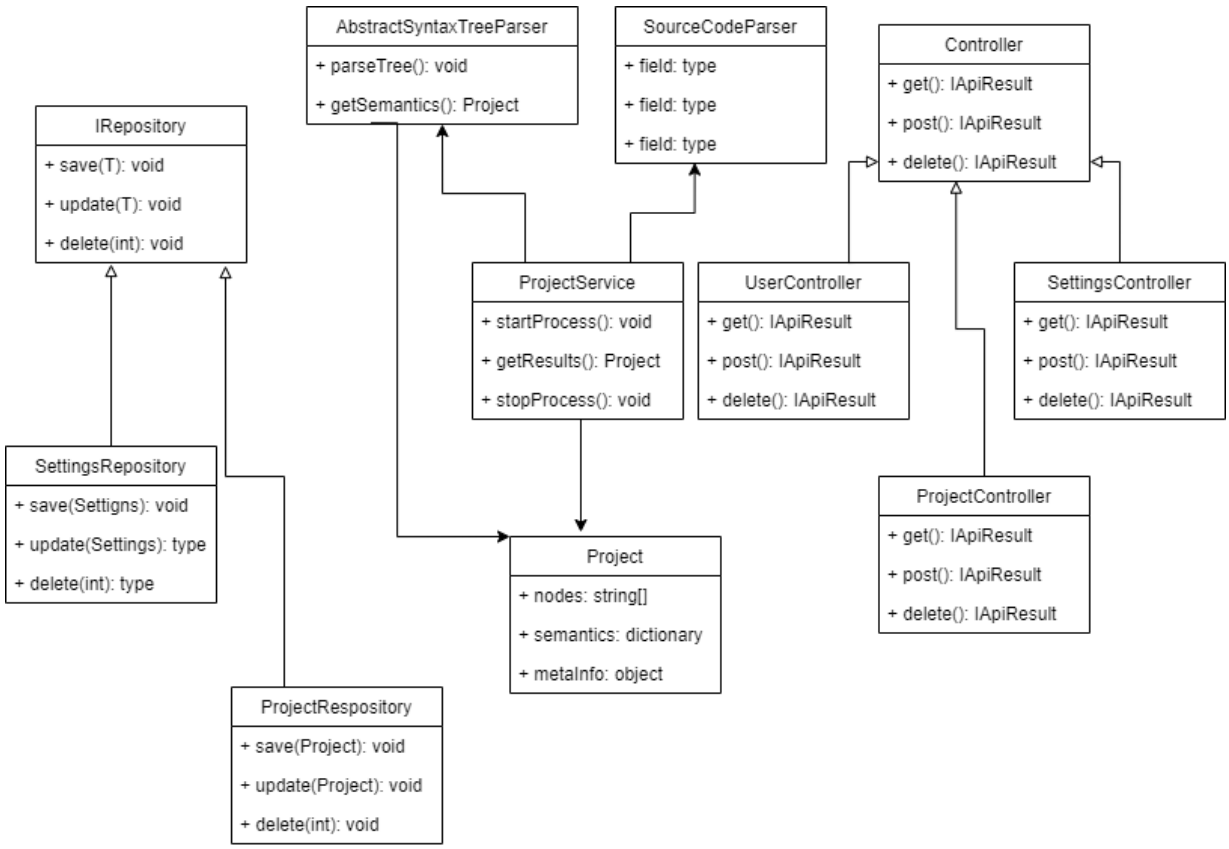
AbstractTreeParser.cs

```
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;
using Microsoft.CodeAnalysis.Diagnostics;
using System.Collections.Immutable;
using System.Linq;

namespace CodeCracker.CSharp.Style
{
    [DiagnosticAnalyzer(LanguageNames.CSharp)]
    public class AlwaysUseVarAnalyzer : DiagnosticAnalyzer
    {
        internal const string Title = "You should use 'var' whenever possible.";
        internal const string MessageFormat = "Use 'var' instead of specifying the
type name.";
        internal const string Category = SupportedCategories.Style;
        const string Description = "Usage of an implicit type improve readability of
the code.\r\n"
            + "Code depending on types for their readability should be refactored
with better variable "
            + "names or by introducing well-named methods.";
        internal static readonly DiagnosticDescriptor RuleNonPrimitives = new
DiagnosticDescriptor(
            DiagnosticId.AlwaysUseVar.ToDiagnosticId(),
            Title,
            MessageFormat,
            Category,
            DiagnosticSeverity.Warning,
            isEnabledByDefault: true,
            description: Description,
            helpLinkUri: HelpLink.ForDiagnostic(DiagnosticId.AlwaysUseVar));

        internal static readonly DiagnosticDescriptor RulePrimitives = new
DiagnosticDescriptor(
            DiagnosticId.AlwaysUseVarOnPrimitives.ToDiagnosticId(),
            Title,
            MessageFormat,
            Category,
            DiagnosticSeverity.Warning,
```

```
            isEnabledByDefault: true,
            description: Description,
            helpLinkUri:
HelpLink.ForDiagnostic(DiagnosticId.AlwaysUseVarOnPrimitives));

        public override ImmutableArray<DiagnosticDescriptor> SupportedDiagnostics =>
            ImmutableArray.Create(RuleNonPrimitives, RulePrimitives);

        public override void Initialize(AnalysisContext context) =>
            context.RegisterSyntaxNodeAction(AnalyzeNode,
SyntaxKind.LocalDeclarationStatement);

        private static void AnalyzeNode(SyntaxNodeAnalysisContext context)
        {
            if (context.IsGenerated()) return;
            var localDeclaration = (LocalDeclarationStatementSyntax)context.Node;
            if (localDeclaration.IsConst) return;

            var variableDeclaration = localDeclaration.ChildNodes()
                .OfType<VariableDeclarationSyntax>()
                .FirstOrDefault();

            if (variableDeclaration.Type.IsVar) return;
            var isDynamic = (variableDeclaration.Type as
IdentifierNameSyntax)?.Identifier.ValueText == "dynamic";

            var semanticModel = context.SemanticModel;
            var variableTypeName = localDeclaration.Declaration.Type;
            var variableType =
semanticModel.GetTypeInfo(variableTypeName).ConvertedType;

            foreach (var variable in variableDeclaration.Variables)
            {
                if (variable.Initializer == null) return;
                var conversion =
semanticModel.ClassifyConversion(variable.Initializer.Value, variableType);
                if (!conversion.IsIdentity) return;
                if (isDynamic)
                {
                    var expressionReturnType =
semanticModel.GetTypeInfo(variable.Initializer.Value);
                    if (expressionReturnType.Type.SpecialType ==
SpecialType.System_Object) return;
                }
            }

            var rule = variableType.IsPrimitive() ? RulePrimitives :
RuleNonPrimitives;
            var diagnostic = Diagnostic.Create(rule,
variableDeclaration.Type.GetLocation());
            context.ReportDiagnostic(diagnostic);
        }
    }
}
```

ДОДАТОК Б

СХЕМА СТРУКТУРНА КЛАСІВ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ


AbstractSyntaxTreeParser
+ parseTree(): void
+ getSemantics(): Project
SourceCodeParser
+ field: type
+ field: type
+ field: type
Controller
+ get(): IApiResult
+ post(): IApiResult
+ delete(): IApiResult
IRepository
+ save(T): void
+ update(T): void
+ delete(int): void
ProjectService
+ startProcess(): void
+ getResults(): Project
+ stopProcess(): void
UserController
+ get(): IApiResult
+ post(): IApiResult
+ delete(): IApiResult
SettingsController
+ get(): IApiResult
+ post(): IApiResult
+ delete(): IApiResult
SettingsRepository
+ save(Settigns): void
+ update(Settings): type
+ delete(int): type
Project
+ nodes: string[]
+ semantics: dictionary
+ metaInfo: object
ProjectController
+ get(): IApiResult
+ post(): IApiResult
+ delete(): IApiResult
ProjectRespository
+ save(Project): void
+ update(Project): void
+ delete(int): void